CONTEXT

General

group: CONTEXT System Macros

version: 1996.3.20

date: 1997 July 25

author: Hans Hagen

The following macros are responsible for the interaction with CONTEXT. These macros have proven their use. These macros are optimized as far as possible within of course, the know how of the author.

In this module we also show some of the optimizations, mainly because we don't want to forget them and start doing things over and over again. If showing them has a learing effect for others too, we've surved another purpose too.

\abortinputi.. Because this module can be used in a different context, we want to prevent it being loaded more than once. This can be done using:

```
\abortinputifdefined\command
```

where \command is a command defined in the module to be loaded only once.

```
\def\abortinputifdefined#1%
  {\ifx#1\undefined
     \let\next=\relax
   \else
     \let\next=\endinput
   \fi
   \next}
```

This macro can be speed up in terms of speed as well as memory. Because this is a nice example of a bit strange command (\endinput), we spend some more lines on this.

If we perform such actions directly, we can say:

```
\ifx\somecommand\undefined
  \let\next=\relax
\else
  \let\next=\endinput
\fi
\next
```

We need the \next because we need to end the \fi. The efficient one is:

```
\ifx\somecommand\undefined
\else
  \expandafter\endinput
\fi
```

Because \endinput comes into action after the current line, we can also say:

```
\ifx\somecommand\undefined \else \endinput \fi
```

When we define a macro, we tend to use a format which shows as besat as can how things are done. TEX however stores the definitions as a sequence of tokens, so in fact we can use a formatted definition:

```
1  \def\abortinputifdefined#1%
     {\ifx#1\undefined \else
        \endinput
      \fi}
```

which also works. Keep in mind that this is entirely due to the fact that \endinput after the line, i.e. at the end of the macro. We therefore can burry this primitive quite deep in code.

And because this module implements \writestatus, we just say:

```
\abortinputifdefined\writestatus
```

Normally we tell the users what module is being loaded. However, the command that is needed for this is not yet defined.

```
\writestatus{laden}{Context Systeem Macro's (a)}
```

\protect
\unprotect

We can shield macros from users by using some special characters in their names. Some characters that are normally no letters and therefore often used are: `@`, `!` and `?`. Before and after the definition of protected macros, we have to change the ⟨*catcode*⟩ of these characters. This is done by `\unprotect` and `\protect`, for instance:

```
\unprotect
\def\!test{test}
\protect
```

The defined command `\!test` can of course only be called upon when we are in the `\unprotect`'ed state, otherwise TEX reads `\!` and probably complains loudly about not being in math mode.

Both commands can be used nested, but only the ⟨*catcode*⟩ of the outermost level is saved. We make use of an auxilary macro `\doprotect` to prevent us from conflicts with existing macro's `\protect`. When nesting deeper than one level, the system shows the protection level.

```
\newcount\protectionlevel
```

```
\ifx\protect\undefined
  \def\protect{\message{<too much protection>}}
\fi
```

```
\let\normalprotect=\protect
```

```
\def\unprotect%
  {\ifnum\protectionlevel=0
     \edef\doprotectcharacters%
       {\catcode`@=\the\catcode`@\relax
        \catcode`!=\the\catcode`!\relax
        \catcode`?=\the\catcode`?\relax}%
     \catcode`@=11
     \catcode`!=11
     \catcode`?=11
     \let\protect=\doprotect
   \fi
   \advance\protectionlevel by 1
   \ifnum\protectionlevel>1
     \message{<unprotect \the\protectionlevel>}%
   \fi}
```

```
\def\doprotect%
  {\ifnum\protectionlevel=1
     \doprotectcharacters
     \let\protect=\normalprotect
   \fi
   \ifnum\protectionlevel>1
     \message{<protect \the\protectionlevel>}%
   \fi
   \advance\protectionlevel by -1\relax}
```

Now it is defined, we can make use of this very useful macro.

```
8  \unprotect
```

\@@escape \@@begingroup \@@endgroup \@@mathshift \@@alignment \@@endofline \@@parameter \@@superscript \@@subscript \@@ignore \@@space \@@letter \@@other \@@active \@@comment

In CONTEXT we sometimes manipulate the ⟨*catcodes*⟩ of certain characters. Because we are not that good at numbers, we introduce some symbolic names.

```
\chardef\@@escape      =  0
\chardef\@@begingroup  =  1
\chardef\@@endgroup    =  2
\chardef\@@mathshift   =  3
\chardef\@@alignment   =  4
\chardef\@@endofline   =  5
\chardef\@@parameter   =  6
\chardef\@@superscript =  7
\chardef\@@subscript   =  8
\chardef\@@ignore      =  9
9  \chardef\@@space       = 10
\chardef\@@letter      = 11
\chardef\@@other       = 12   \chardef\other  = 12
\chardef\@@active      = 13   \chardef\active = 13
\chardef\@@comment     = 14
```

\normalspace

We often need a space as defined in PLAIN TEX. Because we cannot be sure of `\space` is redefined, we define:

```
10  \def\normalspace{ }
```

\scratchcoun.. \scratchdimen \scratchskip \scratchmuskip \scratchbox \scratchtoks..

Because we often need counters on a temporary basis, we define the ⟨*counter*⟩ `\scratchcounter`. This is a real ⟨*counter*⟩, and not a pseudo one, as we will meet further on. We also define some other scratch registers.

```
\newcount  \scratchcounter
\newdimen  \scratchdimen
11  \newskip   \scratchskip
\newmuskip \scratchmuskip
\newbox    \scratchbox
\newtoks   \scratchtoks
\newif     \ifdone
```

\ifCONTEXT

In the system and support modules we sometimes show examples that make use of core commands. We can skip those parts of the documentation when we use another macropackage. Of course we default to false.

```
12  \newif \ifCONTEXT
```

\!!count \!!toks \!!dimen \!!box \!!width \!!height \!!depth \!!string \!!done

We define some more ⟨*counters*⟩ and ⟨*dimensions*⟩. We also define some shortcuts to the local scatchregisters 0, 2, 4, 6 and 8.

```
\newcount\!!counta \toksdef\!!toksa=0 \dimendef\!!dimena=0 \chardef\!!boxa=0
\newcount\!!countb \toksdef\!!toksb=2 \dimendef\!!dimenb=2 \chardef\!!boxb=2
\newcount\!!countc \toksdef\!!toksc=4 \dimendef\!!dimenc=4 \chardef\!!boxc=4
\newcount\!!countd \toksdef\!!toksd=6 \dimendef\!!dimend=6 \chardef\!!boxd=6
\newcount\!!counte \toksdef\!!tokse=8 \dimendef\!!dimene=8 \chardef\!!boxe=8
13 \newcount\!!countf

14 \def\!!stringa{} \def\!!stringb{} \def\!!stringc{}
   \def\!!stringd{} \def\!!stringe{} \def\!!stringf{}

15 \newdimen\!!widtha \newdimen\!!heighta \newdimen\!!deptha \newif\if!!donea
   \newdimen\!!widthb \newdimen\!!heightb \newdimen\!!depthb \newif\if!!doneb
```

\s! \c! \e! \p! \v! \@@ \??

To save memory, we use constants (sometimes called variables). Redefining these constants can have desastrous results.

```
\def\v!prefix! {v!}          \def\c!prefix! {c!}
\def\s!prefix! {s!}          \def\p!prefix! {p!}

\def\s!next    {next}        \def\s!default {default}
\def\s!dummy   {dummy}       \def\s!unknown {unknown}

\def\s!do      {do}          \def\s!dodo    {dodo}

\def\s!complex {complex}     \def\s!start   {start}
19 \def\s!simple  {simple}    \def\s!stop    {stop}
```

\@EA \expanded

When in unprotected mode, to be entered with `\unprotect`, one can use `\@EA` as equivalent of `\expandafter`.

```
20 \let\@EA=\expandafter
```

Sometimes we pass macros as arguments to commands that don't expand them before interpretation. Such commands can be enclosed with `\expanded`, like:

```
\expanded{\setupsomething[\alfa]}
```

Such situations occur for instance when `\alfa` is a commalist or when data stored in macros is fed to index of list commands. If needed, one should use `\noexpand` inside the argument. Later on we will meet some more clever alternatives to this command.

```
21 \def\expanded#1%
     {\edef\@@expanded{\noexpand#1}\@@expanded}
```

The next set of macros just do nothing, except that they get rid of a number of arguments.

\gobbleonear.. \gobble...ar22.

```
\long\def\gobbleoneargument    #1{}
\long\def\gobbletwoarguments   #1#2{}
\long\def\gobblethreearguments #1#2#3{}
\long\def\gobblefourarguments  #1#2#3#4{}
\long\def\gobblefivearguments  #1#2#3#4#5{}
\long\def\gobblesixarguments   #1#2#3#4#5#6{}
\long\def\gobblesevenarguments #1#2#3#4#5#6#7{}
\long\def\gobbleeightarguments #1#2#3#4#5#6#7#8{}
\long\def\gobblenineararguments  #1#2#3#4#5#6#7#8#9{}
```

\doifnextcha..

When we started using TEX in the late eighties, our first experiences with programming concerned a simple shell around LATEX. The commands probably use most at PRAGMA, are the itemizing ones. One of those few shell commands took care of an optional argument, that enabled us to specify what kind of item symbol we wanted. Without understanding anything we were able to locate a LATEX macro that could be used to inspect the next character.

It's this macro that the ancester of the next one presented here. It executes one of two actions, dependant of the next character. Disturbing spaces and line endings, which are normally interpreted as spaces too, are skipped.

```
\doifnextcharelse {karakter} {then ...} {else ...}
```

This macro differs from the original in testing on `\endoflinetoken`, which of course we have to define first. We also use `\localnext` because we don't want clashes with `\next`.

23
```
\let\endoflinetoken=^^M
```

24
```
\long\def\doifnextcharelse#1#2#3%
   {\let\charactertoken=#1%
    \def\!!stringa{#2}%
    \def\!!stringb{#3}%
    \futurelet\nexttoken\inspectnextcharacter}
```

25
```
\def\inspectnextcharacter%
  {\ifx\nexttoken\blankspace
     \let\localnext\reinspectnextcharacter
   \else\ifx\!!stringc\endoflinetoken
     \let\localnext\reinspectnextcharacter
   \else\ifx\nexttoken\charactertoken
     \let\localnext\!!stringa
   \else
     \let\localnext\!!stringb
   \fi\fi\fi
   \localnext}
```

This macro uses some auxiliary macros. Although we were able to program quite complicated things, I only understood these after rereading the TEXbook. The trick is in using a command with a one character name. Such commands differ from the longer ones in the fact that trailing spaces are *not* skipped. This enables us to indirectly define a long named macro that gobbles a space.

In the first line we define `\blankspace`. Next we make `\:` equivalent to `\reinspect...`. This one–character command is expanded before the next `\def` comes into action. This way the space after `\:` becomes a delimiter of the longer named `\reinspectnextcharacter`. The chain reaction is visually compatible with the next sequence:

```
\expandafter\def\reinspectnextcharacter %
  {\futurelet\nexttoken\inspectnextcharacter}
```

However complicated it may look, I'm still glad I stumbled into this construction.

26
```
\def\:{\let\blankspace= }  \:
```

27
```
\def\:{\reinspectnextcharacter}
```

28
```
\expandafter\def\: {\futurelet\nexttoken\inspectnextcharacter}
```

\setvalue
\setgvalue
\setevalue
\setxvalue
\letvalue
\getvalue
\resetvalue

TEX's primitive \csname can be used to construct all kind of commands that cannot be defined with \def and \let. Every macro programmer sooner or later wants macros like these.

```
\setvalue   {naam}{...} = \def\naam{...}
\setgvalue  {naam}{...} = \gdef\naam{...}
\setevalue  {naam}{...} = \edef\naam{...}
\setxvalue  {naam}{...} = \xdef\naam{...}
\letvalue   {naam}=\... = \let\naam=\...
\getvalue   {naam}      = \naam
\resetvalue {naam}      = \def\naam{}
```

As we will see, CONTEXT uses these commands many times, which is mainly due to its object oriented and parameter driven character.

29
```
\def\setvalue#1%
  {\expandafter\def\csname#1\endcsname}
```

30
```
\def\setgvalue#1%
  {\expandafter\gdef\csname#1\endcsname}
```

31
```
\def\setevalue#1%
  {\expandafter\edef\csname#1\endcsname}
```

32
```
\def\setxvalue#1%
  {\expandafter\xdef\csname#1\endcsname}
```

33
```
\def\getvalue#1%
  {\csname#1\endcsname}
```

34
```
\def\letvalue#1%
  {\expandafter\let\csname#1\endcsname}
```

35
```
\def\resetvalue#1%
  {\setvalue{#1}{}}
```

\donottest
\unexpanded

When expansion of a macro gives problems, we can precede it by \donottest. It seems that protection is one of the burdens of developers of packages, so maybe that's why in e-TEX protection will be solved in a more robust way.

Sometimes prefixing the macro with \donottest leads to defining an auxiliary macro, like

```
\def\dosomecommand {... ... ...}
\def\somecommand   {\donottest\dosomecommand}
```

This double definition can be made transparant by using \protecte, as in:

```
\unexpanded\def\somecommand{... ... ...}
```

The protection mechanism uses:

36
```
\def\dontprocesstest#1%
  {==}
```

37
```
\def\doprocesstest#1%
  {#1}
```

38 `\let\donottest=\doprocesstest`

By the way, we use a placeholder because we don't want interference when testing on empty strings. Using a placeholder of 8 characters increases the processing time of simple `\doifelse` tests by about 10 %. When we process the test, we have to remove the braces and therefore explictly gobble `#1`.

The fact that many macros have the same prefix, could have a negative impact on searching in the hash table. Because some simple testing does not show differences, we just use:

```
\def\unexpanded#1#2%
  {\@EA#1\@EA#2\@EA{\@EA\donottest\csname\s!do\string#2\endcsname}%
   \@EA#1\csname\s!do\string#2\endcsname}
```

Well, in fact we use the bit more versaatile alternative:

39
```
\def\dosetunexpanded#1#2%
  {\@EA#1\@EA{\@EA#2\@EA}%
     \@EA{\@EA\donottest\csname\s!do\@EA\string\csname#2\endcsname\endcsname}%
   \@EA#1{\s!do\@EA\string\csname#2\endcsname}}
```

40
```
\def\docomunexpanded#1#2%
  {\@EA#1\@EA#2\@EA{\@EA\donottest\csname\s!do\string#2\endcsname}%
   \@EA#1\csname\s!do\string#2\endcsname}
```

41
```
\def\unexpanded#1%
  {\def\dounexpanded%
     {\ifx\next\bgroup
        \@EA\dosetunexpanded
      \else
        \@EA\docomunexpanded
      \fi#1}%
   \futurelet\next\dounexpanded}
```

This one accepts the more direct `\def` and cousins as well as the CONTEXT specific `\setvalue` ones.

And so the definition in our example turns out to be:

```
\def\csname do\somecommand\endcsname{... ... ...}
\def\somecommand{\donottest\csname do\somecommand\endcsname}
```

In which `do\somecommand` is hidden from the user and cannot lead to confusion. It's still permitted to define auxiliary macros like `\dosomecommand`.

When we are going to use e-TEX, we'll probably end up redefining some commands, but we can probably keep the `\unexpanded` ones unchanged.

`\doifundefined`
`\doifdefined`
`\doifundefin..`
`\doifdefined..`
`\doifalldefi..`

The standard way of testing if a macro is defined is comparing its meaning with another undefined one, usually `\undefined`. To garantee correct working of the next set of macros, `\undefined` may never be defined!

```
\doifundefined      {string}    {...}
\doifdefined        {string}    {...}
\doifundefinedelse  {string}    {then ...} {else ...}
\doifdefinedelse    {string}    {then ...} {else ...}
\doifalldefinedelse {commalist} {then ...} {else ...}
```

Every macroname that TEX builds gets an entry in the hash table, which is of limited size. It is expected that e-TEX will offer a less memory–consuming alternative.

Although it will probably never be a big problem, it is good to be aware of the difference between testing on a macro name to be build by using `\csname` and `\endcsname` and testing the `\name` directly.

```
\expandafter\ifx\csname NameA\endcsname\relax ... \else ... \fi

\ifx\NameB\undefined ... \else ... \fi
```

I became aware of this when I mistakenly testen the first one against `\undefined`. When TEX build a name using `\csname` it automatically sets it to `\relax`, which is definitely not the same as `\undefined`. The quickest way to check these things is asking TEX to show the meaning of the names:

```
\expandafter\show\csname NameA\endcsname

\show\NameB
```

The main reason why this never will be a big problem is that when one uses the `\csname` way, one probably has to do with some macroname that always is dealt with that way. Confusion can however arise when one applies both testing methods to the same macroname. By the way, the assignment of `\relax` obeys grouping.

The first one gets rid of `#1`, but still expands to something and the second one expands to `#1`. Because we accept arguments between `{}`, we have to get rid of one level of braces.

Our first implementation of `\ifundefined` was straightforward and readable:

```
\def\ifundefined#1%
  {\expandafter\ifx\csname#1\endcsname\relax}%

\def\doifundefinedelse#1#2#3%
  {\let\donottest=\dontprocesstest
   \ifundefined{#1}%
     \let\donottest=\doprocesstest#2%
   \else
     \let\donottest=\doprocesstest#3%
   \fi}

\def\doifdefinedelse#1#2#3%
  {\doifundefinedelse{#1}{#3}{#2}}

\def\doifundefined#1#2%
  {\doifundefinedelse{#1}{#2}{}}

\def\doifdefined#1#2%
  {\doifundefinedelse{#1}{}{#2}}

\def\doifalldefinedelse#1#2#3%
  {\bgroup
   \donetrue
   \def\checkcommand##1%
     {\doifundefined{##1}{\donefalse}}%
   \processcommalist[#1]\checkcommand
   \ifdone
```

```
      \egroup#2%
    \else
      \egroup#3%
    \fi}
```

When this module was optimized, timing showed that the next alternative can be upto twice as fast, especially when longer arguments are used.

42
```
\def\ifundefined#1%
  {\expandafter\ifx\csname#1\endcsname\relax}
```

43
```
\def\p!doifundefined#1%
  {\let\donottest=\dontprocesstest
   \expandafter\ifx\csname#1\endcsname\relax}
```

44
```
\def\doifundefinedelse#1#2#3%
  {\p!doifundefined{#1}%
     \let\donottest=\doprocesstest#2%
   \else
     \let\donottest=\doprocesstest#3%
   \fi}
```

45
```
\def\doifdefinedelse#1#2#3%
  {\p!doifundefined{#1}%
     \let\donottest=\doprocesstest#3%
   \else
     \let\donottest=\doprocesstest#2%
   \fi}
```

46
```
\def\doifundefined#1#2%
  {\p!doifundefined{#1}%
     \let\donottest=\doprocesstest#2%
   \else
     \let\donottest=\doprocesstest
   \fi}
```

47
```
\def\doifdefined#1#2%
  {\p!doifundefined{#1}%
     \let\donottest=\doprocesstest
   \else
     \let\donottest=\doprocesstest#2%
   \fi}
```

Before we start using this variant, we used another one, which is even a bit faster. This one looked like:

```
\def\p!doifundefined%
  {\begingroup
   \let\donottest=\dontprocesstest
   \ifundefined}

\def\doifundefinedelse#1#2#3%
  {\p!doifundefined{#1}%
     \endgroup#2%
   \else
```

```
    \endgroup#3%
  \fi}
```

A even more previous version used `\bgroup` and `\egroup`. In math mode however, `$1{x}2$` differs from `$1x2$`. This can been seen when one compares the output of:

```
$\kern10pt\showthe\lastkern$
$\kern10pt{\showthe\lastkern}$
$\kern10pt\begingroup\showthe\lastkern\endgroup$
```

When we were developing the scientific units module, we encountered different behavior in text and math mode, which was due to this grouping subtilities. We therefore decided to use `\begingroup` instead of `\bgroup`. Later, when we had optimized some macro's the grouped solution turned out to be unsafe when typesetting this documentation, especially when using `\globaldefs`.

We still have to define `\doifalldefinedelse`. Watch the use of grouping, which garantees local use of the boolean `\ifdone`.

48
```
\def\docheckonedefined#1%
  {\ifundefined{#1}%
     \donefalse
   \fi}
```

49
```
\def\doifalldefinedelse#1#2#3%
  {\begingroup
   \let\donottest=\dontprocesstest
   \donetrue
   \processcommalist[#1]\docheckonedefined
   \ifdone
     \endgroup\let\donottest=\doprocesstest#2%
   \else
     \endgroup\let\donottest=\doprocesstest#3%
   \fi}
```

`\doif` `\doifelse` `\doifnot` `\donottest`

Programming in TEX differs from programming in procedural languages like MODULA. This means that one — well, let me speek for myself — tries to do the things in the well known way. Therefore the next set of `\ifthenelse` commands were between the first ones we needed. A few years later, the opposite became true: when programming in MODULA, I sometimes miss handy things like grouping, runtime redefinition, expansion etc. While MODULA taught me to structure, TEX taught me to think recursive.

```
\doif     {string1} {string2} {...}
\doifnot  {string1} {string2} {...}
\doifelse {string1} {string2} {then ...}{else ...}
```

When expansion gives problems, we can precede the troublemaker with `\donottest`.

This implementatie does not use the construction which is more robust for nested conditionals.

```
\ifx\!!stringa\!!stringb
  \def\next{#3}%
\else
  \def\next{#4}%
\fi
\next
```

In practice, this alternative is at least 20% slower than the alternative used here. The few cases in which we really need the `\next` construction, often need some other precautions and or adaptions too.

50
```
\long\def\doif#1#2#3%
  {\let\donottest=\dontprocesstest
   \edef\!!stringa{#1}%
   \edef\!!stringb{#2}%
   \let\donottest=\doprocesstest
   \ifx\!!stringa\!!stringb
     #3%
   \fi}
```

51
```
\long\def\doifnot#1#2#3%
  {\let\donottest=\dontprocesstest
   \edef\!!stringa{#1}%
   \edef\!!stringb{#2}%
   \let\donottest=\doprocesstest
   \ifx\!!stringa\!!stringb
   \else
     #3%
   \fi}
```

52
```
\long\def\doifelse#1#2#3#4%
  {\let\donottest=\dontprocesstest
   \edef\!!stringa{#1}%
   \edef\!!stringb{#2}%
   \let\donottest=\doprocesstest
   \ifx\!!stringa\!!stringb
     #3%
   \else
     #4%
   \fi}
```

One could wonder why we don't follow the the same approach as in `\doifdefined` c.s. and use `\begingroup` and `\endgroup`. In this case, this alternative is slower, which is probably due to the fact that more meanings need to be restored.

The in terms of memory more efficient alternative using a auxiliary macro also proved to be slower, so we definitely did not choose for:

```
\def\p!doifelse#1#2%
  {\let\donottest=\dontprocesstest
   \edef\!!stringa{#1}%
   \edef\!!stringb{#2}%
   \let\donottest=\doprocesstest
   \ifx\!!stringa\!!stringb}

\long\def\doif#1#2#3%
  {\p!doifelse{#1}{#2}#3\fi}

\long\def\doifnot#1#2#3%
  {\p!doifelse{#1}{#2}\else#3\fi}

\long\def\doifelse#1#2#3#4%
  {\p!doifelse{#1}{#2}#3\else#4\fi}
```

Optimizations like this are related of course to the bottlenecks in TEX. It seems that restoring saved meanings and passing arguments takes some time.

\doifempty
\doifemptyelse
\doifnotempty

We complete our set of conditionals with:

```
\doifempty     {string} {...}
\doifnot       {string} {...}
\doifemptyelse {string} {then ...} {else ...}
```

This time, the string is not expanded.

53
```
\long\def\doifemptyelse#1#2#3%
  {\def\!!stringa{#1}%
   \ifx\!!stringa\empty
     #2%
   \else
     #3%
   \fi}
```

54
```
\long\def\doifempty#1#2%
  {\def\!!stringa{#1}%
   \ifx\!!stringa\empty
     #2%
   \fi}
```

55
```
\long\def\doifnotempty#1#2%
  {\def\!!stringa{#1}%
   \ifx\!!stringa\empty
   \else
     #2%
   \fi}
```

\doifinset
\doifnotinset
\doifinsetelse

We can check if a string is present in a comma separated set of strings. Depending on the result, some action is taken.

```
\doifinset     {string} {string,...} {...}
\doifnotinset  {string} {string,...} {...}
\doifinsetelse {string} {string,...} {then ...} {else ...}
```

The second argument is the comma separated set of strings.

```
\long\def\doifinsetelse#1#2#3#4%
  {\doifelse{#1}{}
     {#4}
     {\donefalse
      \def\v!checkiteminset##1%
        {\doif{#1}{##1}
           {\donetrue
            \let\v!checkiteminset=\gobbleoneargument}}%
      \processcommalist[#2]\v!checkiteminset
      \ifdone
        #3%
      \else
        #4%
      \fi}}

\long\def\doifinset#1#2#3%
  {\doifinsetelse{#1}{#2}{#3}{}}
```

```
\long\def\doifnotinset#1#2#3%
  {\doifinsetelse{#1}{#2}{}{#3}}
```

Because this macro is called quite often we've spent some time optimizing it. This time, the gain in speed is due to (1) defining an external auxiliary macro, (2) not calling any other macros and (3) minimizing the passing of arguments. The gain in speed is impressive.

56
```
\def\p!dodocheckiteminset#1%
  {\edef\!!stringb{#1}%
   \ifx\!!stringa\!!stringb
     \donetrue
     \let\p!docheckiteminset=\gobbleoneargument
   \fi}
```

57
```
\def\p!doifinsetelse#1#2%
  {\let\donottest=\dontprocesstest
   \donefalse
   \edef\!!stringa{#1}%
   \ifx\!!stringa\empty
   \else
     \let\p!docheckiteminset=\p!dodocheckiteminset
     \processcommalist[#2]\p!docheckiteminset
   \fi
   \let\donottest=\doprocesstest
   \ifdone}
```

58
```
\long\def\doifinsetelse#1#2#3#4%
  {\p!doifinsetelse{#1}{#2}%
     #3%
   \else
     #4%
   \fi}
```

59
```
\long\def\doifinset#1#2#3%
  {\p!doifinsetelse{#1}{#2}%
     #3%
   \fi}
```

60
```
\long\def\doifnotinset#1#2#3%
  {\p!doifinsetelse{#1}{#2}%
   \else
     #3%
   \fi}
```

\doifcommon
\doifnotcommon
\doifcommone..

Probably the most time consuming tests are those that test for overlap in sets of strings.

```
\doifcommon     {string,...} {string,...} {...}
\doifnotcommon  {string,...} {string,...} {...}
\doifcommonelse {string,...} {string,...} {then ...} {else ...}
```

We show the slower alternative first, because it shows us how things are done.

```
\long\def\doifcommonelse#1#2#3#4%
  {\donefalse
   \def\p!docommoncheck##1%
     {\def\p!dodocommoncheck####1%
```

```
          {\doif{####1}{##1}
             {\donetrue
              \def\commalistelement{##1}%
              \let\p!docommoncheck=\gobbleoneargument
              \let\p!dodocommoncheck=\gobbleoneargument}}%
        \processcommalist[#2]\p!dodocommoncheck}%
      \processcommalist[#1]\p!docommoncheck
      \ifdone
        #3%
      \else
        #4%
      \fi}

  \long\def\doifcommon#1#2#3%
    {\doifcommonelse{#1}{#2}{#3}{}}

  \long\def\doifnotcommon#1#2#3%
    {\doifcommonelse{#1}{#2}{}{#3}}
```

The processing time is shortened by getting the auxiliary macro to the outermost level and using less `\edef`'s. Sometimes it makes more sence to define local macro's not only because this way we can be sure that they are not redefined, but also because it shows the dependance. In compiled languages, this is no problem at all. It can even save us bytes and processing time. In interpreted languages like TEX it nearly always slows down processing.

61
```
\def\p!dododocommoncheck#1%
  {\edef\!!stringb{#1}%
   \ifx\!!stringa\!!stringb
     \donetrue
     \let\p!docommoncheck\gobbleoneargument
     \let\p!dodocommoncheck\gobbleoneargument
   \fi}
```

62
```
\def\p!doifcommonelse#1#2%
  {\donefalse
   \let\donottest\dontprocesstest
   \let\p!dodocommoncheck\p!dododocommoncheck
   \def\p!docommoncheck##1%
     {\edef\!!stringa{##1}%
      \def\commalistelement{##1}%
      \processcommalist[#2]\p!dodocommoncheck}%
   \processcommalist[#1]\p!docommoncheck
   \let\donottest\doprocesstest
   \ifdone}
```

63
```
\long\def\doifcommonelse#1#2#3#4%
  {\p!doifcommonelse{#1}{#2}%
     #3%
   \else
     #4%
   \fi}
```

64
```
\long\def\doifcommon#1#2#3%
  {\p!doifcommonelse{#1}{#2}%
```

```
    #3%
  \fi}
```

65
```
\long\def\doifnotcommon#1#2#3%
  {\p!doifcommonelse{#1}{#2}%
   \else
     #3%
   \fi}
```

We've already seen some macros that take care of comma separated lists. Such list can be processed with

\processcomm..
\processcomm..
\processcomm..

```
\processcommalist[string,string,...]\commando
```

The user supplied command \commando receives one argument: the string. This command permits nesting and spaces after commas are skipped. Empty sets are no problem.

```
\def\dosomething#1{(#1)}

\processcommalist [\hbox{$a,b,c,d,e,f$}] \dosomething \par
\processcommalist [{a,b,c,d,e,f}]        \dosomething \par
\processcommalist [{a,b,c},d,e,f]        \dosomething \par
\processcommalist [a,b,{c,d,e},f]        \dosomething \par
\processcommalist [a{b,c},d,e,f]         \dosomething \par
\processcommalist [{a,b}c,d,e,f]         \dosomething \par
\processcommalist []                     \dosomething \par
\processcommalist [{[}]                  \dosomething \par
```

Before we show the result, we present the macro's:

66
```
\newcount\commalevel
```

67
```
\def\dododoprocesscommaitem%
  {\csname\s!next\the\commalevel\endcsname}
```

68
```
\def\dodoprocesscommaitem%
  {\ifx\nexttoken\blankspace
     \let\nextcommaitem\redoprocesscommaitem
  %\else\ifx\nexttoken\endoflinetoken
    %\let\nextcommaitem\redoprocesscommaitem
   \else\ifx\nexttoken]%
     \let\nextcommaitem=\gobbleoneargument
   \else
     \let\nextcommaitem=\dododoprocesscommaitem
   \fi\fi%\fi
   \nextcommaitem}
```

69
```
\def\doprocesscommaitem%
  {\futurelet\nexttoken\dodoprocesscommaitem}
```

70
```
\def\doprocesscommalist#1]#2%
  {\advance\commalevel by 1\relax
   \long\expandafter\def\csname\s!next\the\commalevel\endcsname##1,%
     {#2{##1}\doprocesscommaitem}%
   \doprocesscommaitem#1,]\relax
   \advance\commalevel by -1\relax}
```

Empty arguments are not processed. Empty items (,,) however are treated.

71 \def\docheckcommaitem%
  {\ifx\nexttoken]%
     \let\nextcommaitem=\gobbletwoarguments
   \else
     \let\nextcommaitem=\doprocesscommalist
   \fi
   \nextcommaitem}

72 \def\processcommalist[%
  {\futurelet\nexttoken\docheckcommaitem}

We use the same hack for checking the next character, that we use in \doifnextcharelse.

73 \def\:{\redoprocesscommaitem}

74 \expandafter\def\: {\futurelet\nexttoken\dodoprocesscommaitem}

The previous examples lead to:

(a, b, c, d, e, f)

(a)(b)(c)(d)(e)(f)

(a,b,c)(d)(e)(f)

(a)(b)(c,d,e)(f)

(ab,c)(d)(e)(f)

(a,bc)(d)(e)(f)

([])

When a list is saved in a macro, we can use a construction like:

```
\expandafter\processcommalist\expandafter[\list]\command
```

Such solutions suit most situations, but we wanted a bit more.

```
\processcommacommand[string,\stringset,string]\commando
```

where \stringset is a predefined set, like:

```
\def\first{aap,noot,mies}
\def\second{laatste}

\processcommacommand[\first]\message
\processcommacommand[\first,second,third]\message
\processcommacommand[\first,between,\second]\message
```

Commands that are part of the list are expanded, so the use of this macro has it slimits.

75 \def\processcommacommand[#1]%
  {\edef\commacommand{#1}%
   \toks0=\expandafter{\expandafter[\commacommand]}%
   \expandafter\processcommalist\the\toks0 }

The argument to \command is not delimited. Because we often use [] as delimiters, we also have:

```
\processcommalistwithparameters[string,string,...]\command
```

where \command looks like:

```
\def\command[#1]{... #1 ...}
```

76

```
\def\processcommalistwithparameters[#1]#2%
  {\def\docommand##1{#2[##1]}%
   \processcommalist[#1]\docommand}
```

CONTEXT makes extensive use of a sort of case or switch command. Depending of the presence of one or more provided items, some actions is taken. These macros can be nested without problems.

\processaction
\processfirs..
\processalla..

```
\processaction           [x]     [a=>\a,b=>\b,c=>\c]
\processfirstactioninset [x,y,z] [a=>\a,b=>\b,c=>\c]
\processallactionsinset  [x,y,z] [a=>\a,b=>\b,c=>\c]
```

We can supply both a default action and an action to be undertaken when an unknown value is met:

```
\processallactionsinset
  [x,y,z]
  [      a=>\a,
         b=>\b,
         c=>\c,
   default=>\default,
   unknown=>\unknown{... \commalistelement ...}]
```

When #1 is empty, this macro scans list #2 for the keyword default and executed the related action if present. When #1 is non empty and not in the list, the action related to unknown is executed. Both keywords must be at the end of list #2. Afterwards, the actually found keyword is available in \commalistelement. An advanced example of the use of this macro can be found in PPCHTEX, where we completely rely on TEX for interpreting user supplied keywords like SB, SB1..6, SB125 etc.

Even a quick glance at the macros below show some overlap, which means that more efficient alternatives are possible. Because these macro's are very sensitive to subtle changes, we've decided to present the readable originals first Maybe these these macros look complicated, but this is a direct result of the support of nesting. Protection is only applied in \processaction.

```
\newcount\processlevel

\def\processaction[#1]#2[#3]%
  {\doifelse{#1}{}
     {\def\c!compareprocessaction[##1=>##2]%
        {\edef\!!stringa{##1}%
         \ifx\!!stringa\s!default
           \def\commalistelement{#1}%
           ##2%
         \fi}}
     {\let\donottest=\dontprocesstest
      \edef\!!stringb{#1}%
      \let\donottest=\doprocesstest
      \def\c!compareprocessaction[##1=>##2]%
        {\edef\!!stringa{##1}%
         \ifx\!!stringa\!!stringb
```

```
            \def\commalistelement{#1}%
            ##2%
            \let\c!doprocessaction=\gobbleoneargument
          \else\ifx\!!stringa\s!unknown
            \def\commalistelement{#1}%
            ##2%
          \fi\fi}}%
    \def\c!doprocessaction##1%
      {\c!compareprocessaction[##1]}%
    \processcommalist[#3]\c!doprocessaction}

\def\processfirstactioninset[#1]#2[#3]%
  {\doifelse{#1}{}
     {\processaction[][#3]}
     {\def\c!compareprocessaction[##1=>##2][##3]%
        {\edef\!!stringa{##1}%
         \edef\!!stringb{##3}%
         \ifx\!!stringa\!!stringb
           \def\commalistelement{##3}%
           ##2%
           \let\c!doprocessaction=\gobbleoneargument
           \let\c!dodoprocessaction=\gobbleoneargument
         \else\ifx\!!stringa\s!unknown
           \def\commalistelement{##3}%
           ##2%
         \fi\fi}%
      \def\c!doprocessaction##1%
        {\def\c!dodoprocessaction####1%
           {\c!compareprocessaction[####1][##1]}%
         \processcommalist[#3]\c!dodoprocessaction}%
      \processcommalist[#1]\c!doprocessaction}}

\def\processallactionsinset[#1]#2[#3]%
  {\doifelse{#1}{}
     {\processaction[][#3]}
     {\advance\processlevel by 1\relax
      \def\c!compareprocessaction[##1=>##2][##3]%
        {\edef\!!stringa{##1}%
         \edef\!!stringb{##3}%
         \ifx\!!stringa\!!stringb
           \def\commalistelement{##3}%
           ##2%
           \let\c!dodoprocessaction=\gobbleoneargument
         \else\ifx\!!stringa\s!unknown
           \def\commalistelement{##3}%
           ##2%
         \fi\fi}%
      \setvalue{\s!do\the\processlevel}##1%
        {\def\c!dodoprocessaction####1%
           {\c!compareprocessaction[####1][##1]}%
         \processcommalist[#3]\c!dodoprocessaction}%
      \processcommalist[#1]{\getvalue{\s!do\the\processlevel}}%
      \advance\processlevel by -1\relax}}
```

The gain of speed in the final implementation is around 20%, depending on the application.

```
77  \newcount\processlevel

78  \def\v!compareprocessactionA[#1=>#2]%
      {\edef\!!stringb{#1}%
       \ifx\!!stringb\s!default
         #2%
       \fi}

79  \def\v!compareprocessactionB[#1=>#2]%
      {\expandedaction\!!stringb{#1}%
       \ifx\!!stringa\!!stringb
         \def\commalistelement{#1}%
         #2%
         \let\p!doprocessaction=\gobbleoneargument
       \else
         \edef\!!stringb{#1}%
         \ifx\!!stringb\s!unknown
           \def\commalistelement{#1}%
           #2%
         \fi
       \fi}

80  \def\processaction[#1]#2[#3]%
      {\let\donottest=\dontprocesstest
       \expandedaction\!!stringa{#1}%
       \let\donottest=\doprocesstest
       \ifx\!!stringa\empty
         \let\v!compareprocessaction=\v!compareprocessactionA
       \else
         \let\v!compareprocessaction=\v!compareprocessactionB
       \fi
       \def\p!doprocessaction##1%
         {\v!compareprocessaction[##1]}%
       \processcommalist[#3]\p!doprocessaction
       \expandactions}

81  \def\v!compareprocessactionC[#1=>#2][#3]%
      {\expandedaction\!!stringa{#1}%
       \expandedaction\!!stringb{#3}%
       \ifx\!!stringa\!!stringb
         \def\commalistelement{#3}%
         #2%
         \let\p!doprocessaction=\gobbleoneargument
         \let\p!dodoprocessaction=\gobbleoneargument
       \else
         \edef\!!stringa{#1}%
         \ifx\!!stringa\s!unknown
           \def\commalistelement{#3}%
           #2%
         \fi
       \fi}
```

```
\def\processfirstactioninset[#1]#2[#3]%
  {\expandedaction\!!stringa{#1}%
   \ifx\!!stringa\empty
     \processaction[][#3]%
   \else
     \def\p!doprocessaction##1%
       {\def\p!dodoprocessaction####1%
          {\v!compareprocessactionC[####1][##1]}%
        \processcommalist[#3]\p!dodoprocessaction}%
     \processcommalist[#1]\p!doprocessaction
   \fi
   \expandactions}
```

```
\def\v!compareprocessactionD[#1=>#2][#3]%
  {\expandedaction\!!stringa{#1}%
   \expandedaction\!!stringb{#3}%
   \ifx\!!stringa\!!stringb
     \def\commalistelement{#3}%
     #2%
     \let\p!dodoprocessaction=\gobbleoneargument
   \else
     \edef\!!stringa{#1}%
     \ifx\!!stringa\s!unknown
       \def\commalistelement{#3}%
       #2%
     \fi
   \fi}
```

```
\def\processallactionsinset[#1]#2[#3]%
  {\expandedaction\!!stringa{#1}%
   \ifx\!!stringa\empty
     \processaction[][#3]%
   \else
     \advance\processlevel by 1\relax
     \setvalue{\s!do\the\processlevel}##1%
       {\def\p!dodoprocessaction####1%
          {\v!compareprocessactionD[####1][##1]}%
        \processcommalist[#3]\p!dodoprocessaction}%
     \processcommalist[#1]{\getvalue{\s!do\the\processlevel}}%
     \advance\processlevel by -1\relax
   \fi
   \expandactions}
```

Now what are those expansion commands doing there. Well, sometimes we want to compare actions that may consist off commands (i.e. are no constants). In such occasions we can use the a bit slower alternatives:

\unexpandedp..
\unexpandedp..
\unexpandedp..

```
\def\unexpandedprocessfirstactioninset{\dontexpandactions\processfirstactioninset}
\def\unexpandedprocessaction          {\dontexpandactions\processaction}
\def\unexpandedprocessallactionsinset {\dontexpandactions\processallactionsinset}
```

By default we expand actions:

```
\def\expandactions%
  {\let\expandedaction=\edef}
```

87 `\expandactions`

But when needed we convert the strings to meaningful sequences of characters.

88
```
\def\unexpandedaction#1>{}
```

89
```
\def\noexpandedaction#1#2%
  {\def\convertedargument{#2}%
   \@EA\edef\@EA#1\@EA{\@EA\unexpandedaction\meaning\convertedargument}}
```

90
```
\def\dontexpandactions%
  {\let\expandedaction=\noexpandedaction}
```

Sometimes the action to be undertaken depends on the next character. This macro get this character and puts it in `\firstcharacter`.

\getfirstcha..
\firstcharac..

```
\getfirstcharacter {string}
```

A two step expansion is used to prevent problems with complicated arguments, for instance arguments that consist of two or more expandable tokens.

91
```
\def\dogetfirstcharacter#1#2\\%
  {\def\firstcharacter{#1}}
```

92
```
\def\getfirstcharacter#1%
  {\edef\!!stringa{#1}%
   \expandafter\dogetfirstcharacter\!!stringa\\}
```

We can check for the presence of a substring in a given sequence of characters.

\doifinstrin..

```
\doifinsetelse {substring} {string} {then ...} {else ...}
```

An application of this command can be found further on. Like before, we first show some alternatives, like the one we started with:

```
\long\def\p!doifinstringelse#1#2#3#4%
  {\def\c!doifinstringelse##1#1##2##3\war%
     {\if##2@%
        #4%
      \else
        #3%
      \fi}%
   \c!doifinstringelse#2#1@@\war}

\def\doifinstringelse%
  {\ExpandBothAfter\p!doifinstringelse}
```

After this we came to:

```
\def\p!doifinstringelse#1#2%
  {\def\c!doifinstringelse##1#1##2##3\war%
     {\if##2@}%
   \c!doifinstringelse#2#1@@\war}

\def\doifinstringelse#1#2#3#4%
  {\ExpandBothAfter\p!doifinstringelse{#1}{#2}%
     #4%
```

```
    \else
      #3%
    \fi}
```

And finaly it became:

93
```
\def\v!ifinstringelse#1#2%
  {\def\c!ifinstringelse##1#1##2##3\war%
     {\csname\if##2@iffalse\else iftrue\fi\endcsname}%
   \c!ifinstringelse#2#1@@\war}
```

94
```
\def\ifinstringelse#1#2%
  {\expanded{\v!ifinstringelse{#1}{#2}}}
```

95
```
\long\def\doifinstringelse#1#2#3#4%
  {\ifinstringelse{#1}{#2}%
     #3%
   \else
     #4%
   \fi}
```

\doifnumbere..

The next macro executes a command depending of the outcome of a test on numerals. This is probably one of the fastest test possible, exept from a less robust 10–step `\if`–ladder or some tricky `\lcode` checking.

```
\doifnumberelse {string} {then ...} {else ...}
```

The macro accepts `123`, `abc`, `{}`, `\getal` and `\the\count...`.

96
```
\long\def\doifnumberelse#1#2#3%
  {\getfirstcharacter{#1}%
   \@EA\ifinstringelse\firstcharacter{1234567890}%
     #2%
   \else
     #3%
   \fi}
```

Before we had `\ifinstringelse` available, we used:

```
\def\doifnumberelse#1%
  {\getfirstcharacter{#1}%
   \rawdoifinsetelse{\firstcharacter}{1,2,3,4,5,6,7,8,9,0}}
```

A faster but less fail safe alternative is:

```
\dostepwiserecurse{0}{9}{1}
  {\@EA\uccode\@EA`\recurselevel=1}

\long\def\doifnumberelse#1#2#3%
  {\getfirstcharacter{#1}%
   \@EA\ifnum\@EA\uccode\@EA`\firstcharacter=1
     #2%
   \else
     #3%
   \fi}
```

This one only works when the `\firstcharacter` is indeed a character. Numbers and strings of characters go all right, but arguments like `\relax` let things go wrong.

\makerawcomm..
\rawdoinsete..
\rawprocessc..
\rawprocessa..

Some of the commands mentioned earlier are effective but slow. When one is desperately in need of faster alternatives and when the conditions are predictable safe, the \raw alternatives come into focus. A major drawback is that they do not take \c!constants into account, simply because no expansion is done. This is no problem with \rawprocesscommalist, because this macro does not compare anything. Expandable macros are permitted as search string.

```
\makerawcommalist[string,string,...]\stringlist
\rawdoifinsetelse{string}{string,...}{...}{...}
\rawprocesscommalist[string,string,...]\commando
\rawprocessaction[x][a=>\a,b=>\b,c=>\c]
```

Spaces embedded in the list, for instance after commas, spoil the search process. The gain in speed depends on the length of the argument (the longer the argument, the less we gain).

97
```
\def\makerawcommalist[#1]#2%
  {\def\appendtocommalist##1%
     {\doifelse{#2}{}
        {\edef#2{##1}}
        {\edef#2{#2,##1}}}%
   \def#2{}%
   \processcommalist[#1]\appendtocommalist}
```

98
```
\def\rawprocesscommaitem#1,%
  {\if]#1\else
     \csname\s!next\the\commalevel\endcsname{#1}%
     \expandafter\rawprocesscommaitem
   \fi}
```

99
```
\def\rawprocesscommalist[#1]#2%
  {\advance\commalevel by 1\relax
   \expandafter\let\csname\s!next\the\commalevel\endcsname=#2%
   \expandafter\rawprocesscommaitem#1,],\relax
   \advance\commalevel by -1\relax}
```

100
```
\def\rawdoifinsetelse#1#2%
  {\doifinstringelse{,#1,}{,#2,}}
```

101
```
\def\v!rawprocessaction[#1][#2]%
  {\def\c!rawprocessaction##1,#1=>##2,##3\war%
     {\if##3@\else
        \def\v!processaction{##2}%
      \fi}%
   \c!rawprocessaction,#2,#1=>,@\war}
```

102
```
\def\rawprocessaction[#1]#2[#3]%
  {\edef\!!stringa{#1}%
   \edef\!!stringb{undefined}%
   \let\v!processaction=\!!stringb
   \ifx\!!stringa\empty
     \@EA\v!rawprocessaction\@EA[\s!default][#3]%
   \else
      \expandafter\v!rawprocessaction\expandafter[\!!stringa][#3]%
      \ifx\v!processaction\!!stringb
        \@EA\v!rawprocessaction\@EA[\s!unknown][#3]%
      \fi
```

```
  \fi
  \ifx\v!processaction\!!stringb
  \else
    \v!processaction
  \fi}
```

When we process the list `a,b,c,d,e`, the raw routine takes over 30% less time, when we feed 20+ character strings we gain about 20%. Alternatives which use `\futurelet` perform worse. Part of the speedup is due to the `\let` and `\expandafter` in the test.

\processunex..

When processing commalists, the arguments are expanded. The main reason for doing so lays in the fact that these macros are used for interfacing. The next alternative can be used for

```
\processunexpandedcommalist
  [\alfa\beta,\gamma,\delta\epsilon]
  \handleitem
```

This time nesting is not supported.

103
```
\def\processunexpandedcommaitem#1,%
  {\if]\noexpand#1%
     \let\nextcommaitem=\relax
   \else
     \handleunexpandedcommaitem{#1}%
     \let\nextcommaitem=\processunexpandedcommaitem
   \fi
   \nextcommaitem}
```

104
```
\def\processunexpandedcommalist[#1]#2%
  {\def\handleunexpandedcommaitem{#2}%
   \processunexpandedcommaitem#1,],\relax}
```

Or faster:

105
```
\def\processunexpandedcommaitem#1,%
  {\if]\noexpand#1\else
     \handleunexpandedcommaitem{#1}%
     \expandafter\processunexpandedcommaitem
   \fi}
```

\dosetvalue
\dosetevalue
\docopyvalue
\doresetvalue
\dogetvalue

When we are going to do assignments, we have to take multi–linguality into account. For the moment we keep things simple and single–lingual.

```
\dosetvalue   {label}    {variable}   {value}
\dosetevalue  {label}    {variable}   {value}
\docopyvalue  {to label} {from label} {variable}
\doresetvalue {label}    {variable}
```

These macros are in fact auxiliary ones and are not meant for use outside the assignment macros.

106
```
\def\dosetvalue#1#2% #3
  {\@EA\def\csname#1#2\endcsname} % {#3}}
```

107
```
\def\dosetevalue#1#2% #3
  {\@EA\edef\csname#1#2\endcsname} % {#3}}
```

```
\def\doresetvalue#1#2%
  {\@EA\def\csname#1#2\endcsname{}}
```

```
\def\docopyvalue#1#2#3%
  {\@EA\def\csname#1#3\endcsname{\csname#2#3\endcsname}}
```

\doassign
\undoassign
\doassignempty

Assignments are the backbone of CONTEXT. Abhorred by the concept of style file hacking, we took a considerable effort in building a parameterized system. Unfortunately there is a price to pay in terms of speed. Compared to other packages and taking the functionality of CONTEXT into account, the total size of the format file is still very acceptable. Now how are these assignments done.

Assignments can be realized with:

```
\doassign[label][variable=value]
\undoassign[label][variable=value]
```

and:

```
\doassignempty[label][variable=value]
```

Assignments like \doassign are compatible with:

```
\def\labelvariable{value}
```

We do check for the presence of an = and loudly complain of it's missed. We will redefine this macro later on, when a more advanced message mechanism is implemented.

```
\def\p!doassign#1[#2][#3=#4=#5]%
  {\ifx\empty#3\else  % and definitely not \ifx#3\empty
     \ifx\relax#5%
     \writestatus
       {setup}
       {missing '=' after '#3' in line \the\inputlineno}%
     \else
       #1{#2}{#3}{#4}%
     \fi
   \fi}
```

```
\def\doassign[#1][#2]%
  {\p!doassign\dosetvalue[#1][#2==\relax]}
```

```
\def\doeassign[#1][#2]%
  {\p!doassign\dosetevalue[#1][#2==\relax]}
```

```
\def\undoassign[#1][#2]%
  {\p!doassign\doresetvalue[#1][#2==\relax]}
```

```
\def\doassignempty[#1][#2=#3]%
  {\doifundefined{#1#2}
     {\dosetvalue{#1}{#2}{#3}}}
```

\getparameters
\geteparamet..
\forgetparam..

Using the assignment commands directly is not our ideal of user friendly interfacing, so we take some further steps.

```
\getparameters    [label] [...=...,...=...]
\forgetparameters [label] [...=...,...=...]
```

Again, the label identifies the category a variable belongs to. The second argument can be a comma separated list of assignments.

```
\getparameters
  [demo]
  [alfa=1,
   beta=2]
```

is equivalent to

```
\def\demoalfa{1}
\def\demobeta{2}
```

In the pre–multi–lingual stadium CONTEXT took the next approach. With

```
\def\??demo {@@demo}
\def\!!alfa {alfa}
\def\!!beta {beta}
```

calling

```
\getparameters
  [\??demo]
  [\!!alfa=1,
   \!!beta=2]
```

lead to:

```
\def\@@demoalfa{1}
\def\@@demobeta{2}
```

Because we want to be able to distinguish the `!!` pre–tagged user supplied variables from internal counterparts, we will introduce a slightly different tag in the multi–lingual modules. There we will use `c!` or `v!`, depending on the context.

By calling `\p!doassign` directly, we save ourselves some argument passing and gain some speed. Whatever optimizations we do, this command will always be one of the bigger bottlenecks.

The alternative `\geteparameters` — it's funny to see that this alternative saw the light so lately — can be used to do expanded assigments.

115
```
\def\dogetparameters#1[#2]#3[#4]%
  {\def\p!dogetparameter##1%
     {\p!doassign#1[#2][##1==\relax]}%
   \processcommalist[#4]\p!dogetparameter}
```

116
```
\def\getparameters%
  {\dogetparameters\dosetvalue}
```

117
```
\def\geteparameters%
  {\dogetparameters\dosetevalue}
```

118
```
\def\forgetparameters%
  {\dogetparameters\doresetvalue}
```

119
```
\let\getexpandedparameters=\geteparameters
```

Sometimes we explicitly want variables to default to an empty string, so we welcome:

\getemptypar..

```
\getemptyparameters [label] [...=...,...=...]
```

120
```
\def\getemptyparameters[#1]#2[#3]%
  {\def\p!dogetemptyparameter##1%
     {\doassignempty[#1][##1]}%
   \processcommalist[#3]\p!dogetemptyparameter}
```

Some CONTEXT commands take their default setups from others. All commands that are able to provide backgounds or rules around some content, for instance default to the standard command for ruled boxes. Is situations like this we can use:

\copyparamet..

```
\copyparameters [to-label] [from-label] [name1,name2,...]
```

For instance

```
\copyparameters
  [internal][external]
  [alfa,beta]
```

Leads to:

```
\def\internalalfa {\externalalfa}
\def\internalbeta {\externalbeta}
```

By using \docopyvalue we've prepared this command for use in a multi–lingual environment.

121
```
\def\copyparameters[#1]#2[#3]#4[#5]%
  {\doifnot{#1}{#3}
     {\def\docopyparameter##1%
        {\docopyvalue{#1}{#3}{##1}}%
      \processcommalist[#5]\docopyparameter}}
```

A lot of CONTEXT commands take optional arguments, for instance:

\doifassignm..

```
\dothisorthat[alfa,beta]
\dothisorthat[first=foo,second=bar]
\dothisorthat[alfa,beta][first=foo,second=bar]
```

Although a combined solution is possible, we prefer a seperation. The next command takes care of propper handling of such multi–faced commands.

```
\doifassignmentelse {...} {then ...} {else ...}
```

122
```
\def\doifassignmentelse%
  {\doifinstringelse{=}}
```

\ifparameters
\checkparame..

A slightly different one is \checkparameters, which also checks on the presence of a =.

The boolean \ifparameters can be used afterwards. Combining both in one \if–macro would lead to problems with nested \if's.

```
\checkparameters[argument]
```

123
```
\newif\ifparameters
```

124
```
\def\c!checkparameters#1=#2#3\war%
  {\if#2@\parametersfalse\else\parameterstrue\fi}
```

125
```
\def\checkparameters[#1]%
  {\c!checkparameters#1=@@\war}
```

It's possible to get an element from a commalist or a command representing a commalist.

\getfromcomm..
\getfromcomm..
\commalistel..
\getcommalis..
\getcommacom..

```
\getfromcommalist    [string] [n]
\getfromcommacommand [string,\strings,string,...] [n]
```

The difference betwee the two of them is the same as the difference between `\processcomma...`. The found string is stored in `\commalistelement`.

We can calculate the size of a comma separated list by using:

```
\getcommalistsize    [string,string,...]
\getcommacommandsize [string,\strings,string,...]
```

Afterwards, the length is available in the macro `\commalistsize` (not a ⟨*counter*⟩).

126
```
\def\commalistsize{0}
```

127
```
\def\p!dogetcommalistsize#1[#2]%
  {\scratchcounter=0\relax
   \def\p!dodogetcommalistsize##1%
     {\advance\scratchcounter by 1\relax}%
   #1[#2]\p!dodogetcommalistsize   % was [{#2}]
   \edef\commalistsize{\the\scratchcounter}}
```

128
```
\def\getcommalistsize%
  {\p!dogetcommalistsize\processcommalist}
```

129
```
\def\getcommacommandsize%
  {\p!dogetcommalistsize\processcommacommand}
```

130
```
\def\p!dodogetfromcommalist#1%
  {\advance\scratchcounter by -1\relax
   \ifnum\scratchcounter=0\relax
     \gdef\globalcommalistelement{#1}%
     \def\doprocesscommaitem##1]{}%
   \fi}
```

131
```
\def\p!dogetfromcommalist#1[#2]#3[#4]%
  {\global\let\globalcommalistelement=\empty
   \bgroup
   \scratchcounter=#4\relax
   #1[#2]\p!dodogetfromcommalist
   \egroup
   \let\commalistelement=\globalcommalistelement}
```

132
```
\def\getfromcommalist%
  {\p!dogetfromcommalist\processcommalist}
```

133
```
\def\getfromcommacommand%
  {\p!dogetfromcommalist\processcommacommand}
```

Watertight (and efficient) solutions are hard to find, due to the handling of braces during parameters passing and scanning. Nevertheless:

```
\def\dosomething#1{(#1=\commalistsize) }

\getcommalistsize [\hbox{$a,b,c,d,e,f$}] \dosomething 1
```

```
\getcommalistsize [{a,b,c,d,e,f}]       \dosomething 1
\getcommalistsize [{a,b,c},d,e,f]       \dosomething 4
\getcommalistsize [a,b,{c,d,e},f]       \dosomething 4
\getcommalistsize [a{b,c},d,e,f]        \dosomething 4
\getcommalistsize [{a,b}c,d,e,f]        \dosomething 4
\getcommalistsize []                    \dosomething 0
\getcommalistsize [{[}]                 \dosomething 1
```

reports:

(1=1) (1=6) (4=4) (4=4) (4=4) (4=4) (0=0) (1=1)

When working with delimited arguments, spaces and lineendings can interfere. The next set of macros uses TEX' internal scanner for grabbing everything between arguments.

\dosinglearg..
\dodoublearg..
\dotriplearg..
\doquadruple..
\doquintuple..
\dosixtuplea..

```
\dosingleargument\commando    = \commando[#1]
\dodoubleargument\commando    = \commando[#1][#2]
\dotripleargument\commando    = \commando[#1][#2][#3]
\doquadrupleargument\commando = \commando[#1][#2][#3][#4]
\doquintupleargument\commando = \commando[#1][#2][#3][#4][#5]
\dosixtupleargument\commando  = \commando[#1][#2][#3][#4][#5][#6]
```

These macros are used in the following way:

```
\def\dosetupsomething[#1][#2]%
  {... #1 ... #2 ...}

\def\setupsomething%
  {\dodoubleargument\dosetupsomething}
```

The implementation can be surprisingly simple and needs no further explanation, like:

```
\def\dosingleargument#1[#2]%
  {#1[#2]}
\def\dotripleargument#1[#2]#3[#4]#5[#6]%
  {#1[#2][#4][#6]}
\def\doquintupleargument#1%
  {\def\dodoquintupleargument[##1]##2[##3]##4[##5]##6[##7]##8[##9]%
     {#1[##1][##3][##5][##7][##9]}%
   \dodoquintupleargument}
```

Because TEX accepts 9 arguments at most, we have to use two–step solution when getting five or more arguments.

When developing more and more of the real CONTEXT, we started using some alternatives that provided empty arguments (in fact optional ones) whenever the user failed to supply them. Because this more complicated macros enable us to do some checking, we reimplemented the non–empty ones.

134
```
\def\dosingleargument%
  {\def\expectedarguments{1}%
   \dosingleempty}
```

135
```
\def\dodoubleargument%
  {\def\expectedarguments{2}%
   \dodoubleempty}
```

```
\def\dotripleargument%
  {\def\expectedarguments{3}%
   \dotripleempty}

\def\doquadrupleargument%
  {\def\expectedarguments{4}%
   \doquadrupleempty}

\def\doquintupleargument%
  {\def\expectedarguments{5}%
   \doquintupleempty}

\def\doquintuplargument%
  {\def\expectedarguments{6}%
   \dosixtupleempty}
```

We use some signals for telling the calling macros if all wanted arguments are indeed supplied by the user.

\iffirstagum..
\ifsecondarg..
\ifthirdargu..
\iffourtharg..
\iffifthargu..
\ifsixthargu..

```
\newif\iffirstargument
\newif\ifsecondargument
\newif\ifthirdargument
\newif\iffourthargument
\newif\iffifthargument
\newif\ifsixthargument
```

The empty argument supplying macros mentioned before, look like:

\dosingleempty
\dodoubleempty
\dotripleempty
\doquadruple..
\doquintuple..

```
\dosingleempty    \command
\dodoubleempty    \command
\dotripleempty    \command
\doquadrupleempty \command
\doquintupleempty \command
\dosixtupleempty  \command
```

So `\dodoubleempty` leades to:

```
\command[#1][#2]
\command[#1][]
\command[][]
```

Depending of the generousity of the user. Afterwards one can use the `\if...argument` boolean. For novice: watch the stepwise doubling of `#`'s

```
\def\noexpectedarguments {0}
\def\expectedarguments   {0}

\def\dogetargument#1#2#3#4%
  {\doifnextcharelse{#1}
     {\let\expectedarguments=\noexpectedarguments
      #3\dodogetargument}
     {\ifnum\expectedarguments>\noexpectedarguments
        \writestatus
          {setup}
          {\expectedarguments\space argument(s) expected
           in line \the\inputlineno\space}%
```

```
      \fi
      \let\expectedarguments=\noexpectedarguments
      #4\dodogetargument#1#2}}

143 \def\getsingleempty#1#2#3%
      {\def\dodogetargument%
         {#3}%
       \dogetargument#1#2\firstargumenttrue\firstargumentfalse}

144 \def\getdoubleempty#1#2#3%
      {\def\dodogetargument#1##1#2%
         {\def\dodogetargument%
            {#3#1##1#2}%
          \dogetargument#1#2\secondargumenttrue\secondargumentfalse}%
       \dogetargument#1#2\firstargumenttrue\firstargumentfalse}

145 \def\gettripleempty#1#2#3%
      {\def\dodogetargument#1##1#2%
         {\def\dodogetargument#1####1#2%
            {\def\dodogetargument%
               {#3#1##1#2#1####1#2}%
             \dogetargument#1#2\thirdargumenttrue\thirdargumentfalse}%
          \dogetargument#1#2\secondargumenttrue\secondargumentfalse}%
       \dogetargument#1#2\firstargumenttrue\firstargumentfalse}

146 \def\getquadrupleempty#1#2#3%
      {\def\dodogetargument#1##1#2%
         {\def\dodogetargument#1####1#2%
            {\def\dodogetargument#1########1#2%
               {\def\dodogetargument%
                  {#3#1##1#2#1####1#2#1########1#2}%
                \dogetargument#1#2\fourthargumenttrue\fourthargumentfalse}%
             \dogetargument#1#2\thirdargumenttrue\thirdargumentfalse}%
          \dogetargument#1#2\secondargumenttrue\secondargumentfalse}%
       \dogetargument#1#2\firstargumenttrue\firstargumentfalse}

147 \def\getquintupleempty#1#2#3%
      {\def\dodogetargument#1##1#2%
         {\def\dodogetargument#1####1#2%
            {\def\dodogetargument#1########1#2%
               {\def\dodogetargument#1################1#2%
                  {\def\dodogetargument%
                     {#3#1##1#2#1####1#2#1########1#2#1################1#2}%
                   \dogetargument#1#2\fifthargumenttrue\fifthargumentfalse}%
                \dogetargument#1#2\fourthargumenttrue\fourthargumentfalse}%
             \dogetargument#1#2\thirdargumenttrue\thirdargumentfalse}%
          \dogetargument#1#2\secondargumenttrue\secondargumentfalse}%
       \dogetargument#1#2\firstargumenttrue\firstargumentfalse}

148 \def\getsixtupleempty#1#2#3%
      {\def\dodogetargument#1##1#2%
         {\def\dodogetargument#1####1#2%
            {\def\dodogetargument#1########1#2%
               {\def\dodogetargument#1################1#2%
                  {\def\dodogetargument#1################################1#2%
```

```
                {\def\dodogetargument%
                   {#3#1##1#2#1####1#2#1########1#2#1################1%
                           #2#1##################################1#2}%
                  \dogetargument#1#2\sixthargumenttrue\sixthargumentfalse}%
               \dogetargument#1#2\fifthargumenttrue\fifthargumentfalse}%
            \dogetargument#1#2\fourthargumenttrue\fourthargumentfalse}%
         \dogetargument#1#2\thirdargumenttrue\thirdargumentfalse}%
      \dogetargument#1#2\secondargumenttrue\secondargumentfalse}%
   \dogetargument#1#2\firstargumenttrue\firstargumentfalse}
```

149

```
\def\dosingleempty    {\getsingleempty    []}
\def\dodoubleempty    {\getdoubleempty    []}
\def\dotripleempty    {\gettripleempty    []}
\def\doquadrupleempty {\getquadrupleempty []}
\def\doquintupleempty {\getquintupleempty []}
\def\dosixtupleempty  {\getsixtupleempty  []}
```

These maybe too mysterious macros enable us to handle more than one setup at once.

\dosinglearg..
\dodoublearg..
\dodoubleemp..
\dotriplearg..
\dotripleemp..

```
\dosingleargumentwithset \command[#1]
\dodoubleargumentwithset \command[#1][#2]
\dotripleargumentwithset \command[#1][#2][#3]
\dodoubleemptywithset    \command[#1][#2]
\dotripleemptywithset    \command[#1][#2][#3]
```

The first macro calls `\command[##1]` for each string in the set `#1`. The second one calls for `\commando[##1][#2]` and the third, well one may guess. These commands support constructions like:

```
\def\dodefinesomething[#1][#2]%
  {\getparameters[\??xx#1][#2]}

\def\definesomething%
  {\dodoubleargumentwithset\dodefinesomething}
```

Which accepts calls like:

```
\definesomething[alfa,beta,...][variable=...,...]
```

Now a whole bunch of variables like `\@@xxalfavariable` and `\@@xxbetavariable` is defined.

150

```
\def\dosingleargumentwithset#1%
  {\def\dodosinglewithset[##1]%
     {\def\dododosinglewithset####1%
        {#1[####1]}%
      \processcommalist[##1]\dododosinglewithset}%
   \dosingleargument\dodosinglewithset}%
```

151

```
\def\dodoublewithset#1#2%
  {\def\dododoublewithset[##1][##2]%
     {\doifnot{##1}{}
        {\def\dododododoublewithset####1%
           {#2[####1][##2]}%
         \processcommalist[##1]\dododododoublewithset}}%
   #1\dododoublewithset}%
```

```
\def\dodoubleemptywithset%
  {\dodoublewithset\dodoubleempty}

\def\dodoubleargumentwithset%
  {\dodoublewithset\dodoubleargument}

\def\dotriplewithset#1#2%
  {\def\dodotriplewithset[##1][##2][##3]%
     {\doifnot{##1}{}
        {\def\dododotriplewithset####1%
           {#2[####1][##2][##3]}%
         \processcommalist[##1]\dododotriplewithset}}%
   #1\dodotriplewithset}%

\def\dotripleemptywithset%
  {\dotriplewithset\dotripleempty}

\def\dotripleargumentwithset%
  {\dotriplewithset\dotripleargument}
```

\complexorsi..
\complexorsi..

Setups can be optional. A command expecting a setup is prefixed by `\complex`, a command without one gets the prefix `\simple`. Commands like this can be defined by:

```
\complexorsimple {command}
```

When `\command` is followed by a `[setup]`, then

```
\complexcommand [setup]
```

executes, else we get

```
\simplecommand
```

An alternative for `\complexorsimple` is:

```
\complexorsimpleempty {command}
```

Depending on the presence of `[setup]`, this one leads to one of:

```
\complexcommando [setup]
\complexcommando []
```

Many CONTEXT commands started as complex or simple ones, but changed into more versatile (more object oriented) ones using the `\get..argument` commands.

```
\def\complexorsimple#1%
  {\doifnextcharelse{[}
     {\firstargumenttrue\getvalue{\s!complex#1}}
     {\firstargumentfalse\getvalue{\s!simple#1}}}

\def\complexorsimpleempty#1%
  {\doifnextcharelse{[}
     {\firstargumenttrue\getvalue{\s!complex#1}}
     {\firstargumentfalse\getvalue{\s!complex#1}[]}}
```

The previous commands are used that often that we found it worthwile to offer two more alternatives.

\definecompl..
\definecompl.. 159
```
\def\setnameofcommand#1%
  {\bgroup
   \escapechar=-1\relax
   \xdef\nameofcommand{\string#1}%
   \egroup}
```

160
```
\def\definewithnameofcommand#1#2% watch the \donottest
  {\setnameofcommand{#2}%
   \@EA\def\@EA#2\@EA{\@EA\donottest\@EA#1\@EA{\nameofcommand}}}
```

161
```
\def\definecomplexorsimple%
  {\definewithnameofcommand\complexorsimple}
```

162
```
\def\definecomplexorsimpleempty%
  {\definewithnameofcommand\complexorsimpleempty}
```

These commands are called as:

```
\definecomplexorsimple\command
```

Of course, we must have available

```
\def\complexcommand[#1]{...}
\def\simplecommand     {...}
```

Using this construction saves a few string now and then.

\definestart..
Those who get the creeps of expansion may skip the next one. It's one of the most recent additions and concerns `\start-\stop` pairs with complicated arguments.

We won't go into details here, but the general form of this using this command is:

```
\definestartstopcommand\somecommand\v!specifier{arg}{arg}%
  {do something with arg}
```

This expands to something like:

```
\def\somecommand arg \startspecifier arg \stopspecifier%
  {do something with arg}
```

The argumentss can be anything reasonable, but double `#`'s are needed in the specification part, like:

```
\definestartstopcommand\somecommand\v!specifier{[##1][##2]}{##3}%
  {do #1 something #2 with #3 arg}
```

which becomes:

```
\def\somecommand[#1][#2]\startspecifier#3\stopspecifier%
  {do #1 something #2 with #3 arg}
```

We will see some real applications of this command in the core modules.

163
```
\def\definestartstopcommand#1#2#3#4%
  {\def\!stringa{#3}%
   \def\!stringb{\e!start#2}%
   \def\!stringc{#4}%
   \def\!stringd{\e!stop#2}%
```

```
\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA
\def\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA\@EA
#1\@EA\@EA\@EA\@EA\@EA\@EA\@EA
\!stringa\@EA\@EA\@EA
\csname\@EA\@EA\@EA\!stringb\@EA\@EA\@EA\endcsname\@EA
\!stringc
\csname\!stringd\endcsname}
```

\dosinglegro..
\dodoublegro..
\dotriplegro..

We've already seen some commands that take care of optional arguments between `[]`. The next two commands handle the ones with `{}`. They are called as:

```
\dosinglegroupempty \IneedONEargument
\dodoublegroupempty \IneedTWOarguments
\dotriplegroupempty \IneedTREEarguments
```

where `\IneedONEargument` takes one and the others two and three arguments. These macro's were first needed in PPCHTEX.

164

```
\def\dogetgroupargument#1#2%
  {\def\nextnext%
     {\ifx\next\bgroup
        \let\expectedarguments=\noexpectedarguments
        \def\next{#1\dodogetargument}%
      \else\ifx\next\lineending
        \def\next{\bgroup\def\\ {\egroup\dogetgroupargument#1#2}\\}%
      \else\ifx\next\blankspace
        \def\next{\bgroup\def\\ {\egroup\dogetgroupargument#1#2}\\}%
      \else
        \ifnum\expectedarguments>\noexpectedarguments
          \writestatus
            {setup}
            {\expectedarguments\space argument(s) expected
             in line \the\inputlineno\space}%
        \fi
        \let\expectedarguments=\noexpectedarguments
        \def\next{#2\dodogetargument{}}%
      \fi\fi\fi
      \next}%
   \futurelet\next\nextnext}
```

165

```
\def\dosinglegroupempty#1%
  {\def\dodogetargument%
     {#1}%
   \dogetgroupargument\firstargumenttrue\firstargumentfalse}
```

166

```
\def\dodoublegroupempty#1%
  {\def\dodogetargument##1%
     {\def\dodogetargument%
        {#1{##1}}%
      \dogetgroupargument\secondargumenttrue\secondargumentfalse}%
   \dogetgroupargument\firstargumenttrue\firstargumentfalse}
```

167

```
\def\dotriplegroupempty#1%
  {\def\dodogetargument##1%
     {\def\dodogetargument####1%
```

```
      {\def\dodogetargument%
         {#1{##1}{####1}}%
       \dogetgroupargument\thirdargumenttrue\thirdargumentfalse}%
    \dogetgroupargument\secondargumenttrue\secondargumentfalse}%
 \dogetgroupargument\firstargumenttrue\firstargumentfalse}
```

These macros explictly take care of spaces, which means that the next definition and calls are valid:

```
\def\test#1#2#3{[#1#2#3]}

\dotriplegroupempty\test {a}{b}{c}
\dotriplegroupempty\test {a}{b}
\dotriplegroupempty\test {a}
\dotriplegroupempty\test
\dotriplegroupempty\test {a} {b} {c}
\dotriplegroupempty\test {a} {b}
\dotriplegroupempty\test
  {a}
  {b}
```

And alike.

`\wait` The next macro hardly needs explanation. Because no nesting is to be expected, we can reuse `\wait` within `\wait` itself.

168
```
\def\wait%
  {\bgroup
   \read16 to \wait
   \egroup}
```

`\writestring` `\writeline` `\writestatus` `\statuswidth` Maybe one didn't notice, but we've already introduced a macro for showing messages. In the multi–lingual modules, we will also introduce a mechanism for message passing. For the moment we stick to the core macros:

```
\writestring {string}
\writeline
\writestatus {category} {message}
```

Messages are formatted. One can provide the maximum with of the identification string with the macro `\statuswidth`.

169
```
\def\statuswidth {15}
```

170
```
\def\writestring%
  {\immediate\write16}
```

171
```
\def\writeline%
  {\writestring{}}
```

172
```
\def\dosplitstatus#1#2\end%
  {\ifx#1?%
     \loop
       \advance\scratchcounter by 1
       \ifnum\scratchcounter<\statuswidth\relax
         \edef\messagecontentA{\messagecontentA\space}%
     \repeat
```

```
  \else
    \advance\scratchcounter by 1
    \ifnum\scratchcounter<\statuswidth\relax
      \edef\messagecontentA{\messagecontentA#1}%
    \fi
    \dosplitstatus#2\end
  \fi}
```

173
```
\def\writestatus#1#2%
  {\bgroup
   \edef\messagecontentA{}%
   \edef\messagecontentB{#2}%  maybe it's \the\scratchcounter
   \scratchcounter=0
   \expandafter\dosplitstatus#1?\end
   \writestring{\messagecontentA\space:\space\messagecontentB}%
   \egroup}
```

`\debuggerinfo` For debugging purposes we can enhance macros with the next alternative. Here `debuggerinfo` stands for both a macro accepting two arguments and a boolean (in fact a few macro's too).

174
```
\newif\ifdebuggerinfo
```

175
```
\def\debuggerinfo#1#2%
  {\ifdebuggerinfo
     \writestatus{debugger}{#1:: #2}%
   \fi}
```

Finally we do what from now on will be done at the top of the files: we tell the user what we are loading.

176
```
\writestatus{loading}{Context System Macros / General}
```

Well, the real final command is the one that resets the unprotected characters `@`, `?` and `!`.

177
```
\protect
```