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 learning effect for others too, we’ve served another purpose too.

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:

\begin{verbatim}
\def\abortinputifdefined#1{\ifx#1\undefined \else \endinput \fi}
\end{verbatim}

where \texttt{\textbackslash command} is a command defined in the module to be loaded only once.

If we perform such actions directly, we can say:

\begin{verbatim}
\ifx\somecommand\undefined \else \endinput \fi
\end{verbatim}

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

\begin{verbatim}
\ifx\somecommand\undefined \else \expandafter\endinput \fi
\end{verbatim}

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

\begin{verbatim}
\ifx\somecommand\undefined \else \endinput \fi
\end{verbatim}

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:

\begin{verbatim}
\def\abortinputifdefined#1{\ifx#1\undefined \else \endinput \fi}
\end{verbatim}

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

And because this module implements \texttt{\textbackslash writestatus}, we just say:
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 System Macro's (a)}

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 auxiliary 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
4 \ife\protect\undefined
   \def\protect{\message{<too much protection>}}
\fi
5 \let\normalprotect=\protect
6 \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}
7 \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
\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
\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{ }

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

\ifCONTEXT
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.

11 \ifCONTEXT
\newif \ifdone

\newif \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
We define some more \textit{(counters)} and \textit{(dimensions)}. We also define some shortcuts to the local scratch registers \(0, 2, 4, 6\), and \(8\).

\begin{verbatim}
\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
\newcount\!!countf
\def\!!stringa{} \def\!!stringb{} \def\!!stringc{}
\def\!!stringd{} \def\!!stringe{} \def\!!stringf{}
\newdimen\!!widtha \newdimen\!!heighta \newdimen\!!deptha \newif\if!!donea
\newdimen\!!widthb \newdimen\!!heightb \newdimen\!!depthb \newif\if!!doneb
\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}
\def\s!simple {simple} \def\s!stop {stop}
\let\@EA=\expandafter
\@EA
\expanded
\long\def\gobbleoneargument #1{}
\long\def\gobbletwoarguments #1#2{}
\long\def\gobblethreearguments #1#2#3{}
\long\def\gobblefourarguments #1#2#3#4{}
\long\def\gobbelfivearguments #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\gobbleseightarguments #1#2#3#4#5#6#7#8{}
\long\def\gobbleninearguments #1#2#3#4#5#6#7#8#9{}
\end{verbatim}

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

\begin{verbatim}
\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}
\def\s!simple {simple} \def\s!stop {stop}
\let\@EA=\expandafter
\@EA
\expanded
\long\def\gobbleoneargument #1{}
\long\def\gobbletwoarguments #1#2{}
\long\def\gobblethreearguments #1#2#3{}
\long\def\gobblefourarguments #1#2#3#4{}
\long\def\gobbelfivearguments #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\gobbleseightarguments #1#2#3#4#5#6#7#8{}
\long\def\gobbleninearguments #1#2#3#4#5#6#7#8#9{}
\end{verbatim}

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

\begin{verbatim}
\@EA\@EA
\long\def\gobbleoneargument #1{}
\long\def\gobbletwoarguments #1#2{}
\long\def\gobblethreearguments #1#2#3{}
\long\def\gobblefourarguments #1#2#3#4{}
\long\def\gobbelfivearguments #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\gobbleseightarguments #1#2#3#4#5#6#7#8{}
\long\def\gobbleninearguments #1#2#3#4#5#6#7#8#9{}
\end{verbatim}

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

\begin{verbatim}
\long\def\gobbleoneargument #1{}
\long\def\gobbletwoarguments #1#2{}
\long\def\gobblethreearguments #1#2#3{}
\long\def\gobblefourarguments #1#2#3#4{}
\long\def\gobbelfivearguments #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\gobbleseightarguments #1#2#3#4#5#6#7#8{}
\long\def\gobbleninearguments #1#2#3#4#5#6#7#8#9{}
\end{verbatim}
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 ancestor 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.

\inputdocifnextcharelse {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.

\let\endoflinetoken=\cr
\long\def\docifnextcharelse#1#2#3% {
  \let\charactertoken=#1\%
  \def\!!stringa{#2}\%
  \def\!!stringb{#3}\%
  \futurelet\nexttoken\inspectnextcharacter}

\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
  \futurelet\nexttoken\inspectnextcharacter}

This macro uses some auxiliary macros. Although we were able to program quite complicated things, I only understood these after re-reading 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.

\def\:{\let\blankspace= } \:
\def\:\{\reinspectnextcharacter}
\expandafter\def\:\{\futurelet\nexttoken\inspectnextcharacter}
"setvalue" \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, CON\TEX uses these commands many times, which is mainly due to its object oriented and parameter driven character.

\donottest

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 \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 transparent by using \protecte, as in:

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

The protection mechanism uses:

\dontprocesstest
\unexpanded
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 explicitly 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{\@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 versatile alternative:

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

This one accepts the more direct \def and cousins as well as the CONT\TeX\ 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.

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

\doifundefined {string} {...} \doifdefine {string} {...} \doifundefinedelse {string} {then ...} {else ...} \doifdefinedelse {string} {then ...} {else ...} \doifalldefinedelse {commaclist} {then ...} {else ...}
Every macro name 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
\expandafter\ifx\NameB\undefined ... \else ... \fi

I became aware of this when I mistakenly tested 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 macro name that always is dealt with that way. Confusion can however arise when one applies both testing methods to the same macro name. 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}{}}
\def\doifalldefinedelse#1#2#3\%
{\bgroup\donetrue\def\checkcommand##1\%
{\doifundefined{##1}{\donefalse}}\%
\processcommalist[#1]\checkcommand\ifdone
When this module was optimized, timing showed that the next alternative can be up to twice as fast, especially when longer arguments are used.

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

\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:

$\kern10pt\showthe\lastkern$

$\kern10pt\showthe\lastkern$

$\kern10pt\showthe\lastkern$

When we were developing the scientific units module, we encountered different behavior in text and math mode, which was due to this grouping subtleties. 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.

\newcount\doifcheckonedefined
\doifcheckonedefined#1%
{\ifundefined{#1}%
 \donefalse
 \fi}

\newcount\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}

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.

\if\!!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.
One could wonder why we don’t follow 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:

\long\def\doifelse#1#2#3#4%\
\{\let\donottest=\dontprocesstest\
  \edef\!!stringa{#1}\
  \edef\!!stringb{#2}\
  \let\donottest=\doprocesstest\
  \ifx\!!stringa\!!stringb\
    \else\
    \fi\
  \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.
We complete our set of conditionals with:

\doifempty {string} {...}
\doifnotempty {string} {...}
\doifemptyelse {string} {then ...} {else ...}

This time, the string is not expanded.

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

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

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

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.
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.

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

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

```latex
\long\def\doifcommonelse#1#2#3#4\%
{\p\doifinsetelse{#1}{#2}{#3}\%
#4\%}
```

```
\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.
The processing time is shortened by getting the auxiliary macro to the outermost level and using less \edef's. Sometimes it makes more sense 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.
We’ve already seen some macros that take care of comma separated lists. Such lists can be processed with

\processcommalist \string, \string, \ldots \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:

\def \dododo \processcommalist
\def \dodoprocesscommaitem
\def \dodoprocesscommalist #1 \processcommalist #2

\def \dodoprocesscommaitem
\def \dodoprocesscommalist #1 \processcommalist #2

\def \dodoprocesscommalist #1 #2 %
{\futurelet \nexttoken \dodoprocesscommalist #1 \relax
 \long\expandafter \def \csname \s !next \the \commalevel \endcsname \endcsname
 \ifx \nexttoken \blankspacenexttoken \redoprocesscommaitem
 %\else %\nexttoken \endoflinetoken %\let \nextcommaitem \redoprocesscommaitem
 \else \fi
 \advance \commalevel by -1 \relax
 \def \nextcommaitem \gobbleoneargument
 \else \fi
 \let \nextcommaitem \dodoprocesscommaitem
 \fi \fi \fi

\def \dodoprocesscommalist #1
{\futurelet \nexttoken \dodoprocesscommalist #1}

\def \doprocesscommalist #1
{\futurelet \nexttoken \dodoprocesscommalist #1}

\def \doprocesscommalist #1 #2 %
{\futurelet \nexttoken \dodoprocesscommalist #1 \relax
 \long\expandafter \def \csname \s !next \the \commalevel \endcsname \endcsname
 \ifx \nexttoken \blankspacenexttoken \redoprocesscommaitem
 %\else %\nexttoken \endoflinetoken %\let \nextcommaitem \redoprocesscommaitem
 \else \fi
 \advance \commalevel by -1 \relax
 \def \nextcommaitem \gobbleoneargument
 \else \fi
 \let \nextcommaitem \dodoprocesscommalist
 \fi \fi \fi

\def \dodoprocesscommalist #1
{\futurelet \nexttoken \dodoprocesscommalist #1}

\def \doprocesscommalist #1
{\futurelet \nexttoken \dodoprocesscommalist #1}

\def \doprocesscommalist #1 #2 %
{\futurelet \nexttoken \dodoprocesscommalist #1 \relax
 \long\expandafter \def \csname \s !next \the \commalevel \endcsname \endcsname
 \ifx \nexttoken \blankspacenexttoken \redoprocesscommaitem
 %\else %\nexttoken \endoflinetoken %\let \nextcommaitem \redoprocesscommaitem
 \else \fi
 \advance \commalevel by -1 \relax
 \def \nextcommaitem \gobbleoneargument
 \else \fi
 \let \nextcommaitem \dodoprocesscommalist
 \fi \fi \fi

syst-gen CONTEXT CONTEXT System Macros
Empty arguments are not processed. Empty items (,,) however are treated.

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

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

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

\def\: {\redoprocesscommaitem}
\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.

\def\processcommacommand[#1]% 
  {\edef\commacommand{#1}% 
    \toks0=\expandafter{\expandafter[\commacommand]\message
    \expandafter\processcommalist\the\toks0 }
The argument to \texttt{\command} is not delimited. Because we often use \texttt{[]} as delimiters, we also have:

\texttt{\processcommalistwithparameters[string,string,...]\\command}

where \texttt{\command} looks like:

\begin{verbatim}
def\command[#1]{... #1 ...}
\end{verbatim}

\begin{verbatim}
def\processcommalistwithparameters[#1]#2\%
  {\def\docommand##1{#2[##1]}%
    \processcommalist[#1]\docommand}
\end{verbatim}

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

\begin{verbatim}
\processaction \processfirstactioninset \processallactionsinset
\end{verbatim}

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

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

When \texttt{#1} is empty, this macro scans list \texttt{#2} for the keyword \texttt{default} and executed the related action if present. When \texttt{#1} is non empty and not in the list, the action related to \texttt{unknown} is executed. Both keywords must be at the end of list \texttt{#2}. Afterwards, the actually found keyword is available in \texttt{\commalistelement}. An advanced example of the use of this macro can be found in \texttt{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 \texttt{\processaction}.

\begin{verbatim}
newcount\processlevel
\end{verbatim}

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

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

```
\newcount\processlevel

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

\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}

\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}

\def\v!compareprocessactionC[#1=>#2][#3]\
 {\expandedaction!!stringa{#1} \\
   \expandedaction!!stringb{#3} \\
   \ifx!!stringa!!stringb \\
     \def\commalistelement{#3} \\
   \let\p!doprocessaction=\gobbleoneargument \\
   \let\p!doprocessaction=\gobbleoneargument \\
   \else \\
     \edef!!stringa{#1} \\
     \ifx!!stringa\s!unknown \\
       \def\commalistelement{#3} \\
     \fi
   \fi}
```
\def\processfirstactioninset[#1]#2[#3]{% 
  \ifx\!!stringa\empty \processaction[][#3]% 
  \else \def\p!doprocessaction##1% {\def\p!dodoprocessaction####1% {\v!compareprocessactionC[####1][##1]}% \processcommalist[#3]\p!doprocessaction} \processcommalist[#1]\p!dodoprocessaction \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 of commands (i.e. are no constants). In such occasions we can use the a bit slower alternatives:

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

By default we expand actions:

\def\expandactions{% 
  \let\expandedaction=\edef}
But when needed we convert the strings to meaningful sequences of characters.

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

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

\firstcharacter {string}

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

\getfirstcharacter {string}

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

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

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

\doifinstrinelse {substring} {string} {then ...} {else ...}

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

After this we came to:

After this we came to:
And finally it became:

\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}

\def\ifinstringelse#1#2{\expanded{\v!ifinstringelse{#1}{#2}}}

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

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

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

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

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

Before we had \texttt{\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 \texttt{\firstcharacter} is indeed a character. Numbers and strings of characters go all right, but arguments like \texttt{relax} let things go wrong.
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 \texttt{\raw} alternatives come into focus. A major drawback is that they do not take \texttt{\c!constants} into account, simply because no expansion is done. This is no problem with \texttt{\rawprocesscommalist}, because this macro does not compare anything. Expandable macros are permitted as search string.

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

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).

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

98 \def\rawprocesscommalist#1,#2\%
 {\if\]#1\else
 {\csname s!next\the\commalevel\endcsname{#1}%
 \expandafter\rawprocesscommalist
 \fi}

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

100 \def\rawdoifinsetelse#1#2\%
 {\doifstringelse{#1}{#2}}

101 \def\v!rawprocessaction[#1]#2\%
 {\def\c!rawprocessaction[#1]#1=>#2,\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
\end{verbatim}
When we process the list \texttt{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 \texttt{\futurelet} perform worse. Part of the speedup is due to the \texttt{\let} and \texttt{\expandafter} in the test.

When processing comma lists, 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

\begin{verbatim}
\processunexpandedcommalist
  [\alpha\beta,\gamma,\delta\epsilon]
\handleitem
\end{verbatim}

This time nesting is not supported.

\begin{verbatim}
\def\processunexpandedcommalist[#1]#2\%
  \def\handleunexpandedcommaitem{#2}\%
  \processunexpandedcommaitem#1,,]%,\relax
\end{verbatim}

Or faster:

\begin{verbatim}
\def\processunexpandedcommalist[#1]#2\%
  \def\handleunexpandedcommaitem{#1}\%
  \expandafter\processunexpandedcommalist
\fi
\end{verbatim}

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.

\begin{verbatim}
\dosetvalue {label} {variable} {value}
\dosetvalue {label} {variable} {value}
\docopyvalue {to label} {from label} {variable}
\dosetvalue {label} {variable}
\end{verbatim}

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

\begin{verbatim}
\def\dosetvalue#1#2% #3 \% \def\csname#1#2\endcsname% {#3}
\def\dosetvalue#1#2% #3 \% \edef\csname#1#2\endcsname% {#3}
\end{verbatim}
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:
\begin{verbatim}
\doassign[label][variable=value]
\undoassign[label][variable=value]
\doassignempty[label][variable=value]
\end{verbatim}

Assignments like \doassign are compatible with:
\begin{verbatim}
def\labelvariable{value}
\end{verbatim}
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.
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 assignments.

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

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

117 \def\getparameters\
    {\dogetparameters\dosetvalue}

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

119 \let\getexpandedparameters=\getparameters
Sometimes we explicitly want variables to default to an empty string, so we welcome:

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

\def\getemptyparameters[#1]#2[#3]%
  \docopyparameters[...=,...=,...]

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. In situations like this we can use:

\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.

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

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

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

\def\doifassignmentelse%
  {
doifinstringelse={}}

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]
It’s possible to get an element from a comma list or a command representing a comma list.

The difference between 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:

\begin{verbatim}
\commalistsize [string,string,...]
\commacommandsize [string,\strings,string,...]
\end{verbatim}

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

\begin{verbatim}
def\commalistsize{0}
def\getcommalistsize#1[#2]%=\scratchcounter=0\relax
\def\p!dodogetcommalistsize##1%{
\advance\scratchcounter by 1\relax
#1[#2]\p!dodogetcommalistsize % was [{#2}]
def\commalistsize{the}[\scratchcounter]}
def\getcommalistsize\processcommalist
\def\getcommacommandsize\processcommacommand
def\p!dodogetfromcommalist#1=%.\scratchcounter=-1\relax
\ifnum\scratchcounter=0\relax
\gdef\globalcommalistelement{#1}\
def\processcommaiment#1{}\fi}
def\p!dodogetfromcommalist#1[#2]#3[#4]%=\global\let\globalcommalistelement=\empty
\bgroup
\scratchcounter=#4\relax
#1[#2]\p!dodogetfromcommalist
\egroup
\let\commalistelement=\globalcommalistelement}
def\getfromcommalist\processcommalist
def\getfromcommacommand\processcommacommand
\end{verbatim}

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

\begin{verbatim}
def\dosomething#1={(#1=\commalistsize) }
def\commalistsize \hbox{$a,b,c,d,e,f$}\ dosomething 1
\end{verbatim}
When working with delimited arguments, spaces and lineendings can interfere. The next set of macros uses \TeX's internal scanner for grabbing everything between arguments.

These macros are used in the following way:

```latex
\def\dodoubleargument\commando\[#1\][#2]\% 
  {\def\expectedarguments{2} \dodoubleempty}
\def\setup{}\{\dodoubleargument\dosetupsomething\}
```

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

```latex
\def\dodoubleargument\commando\[#1\][#2]\% 
  {\def\expectedarguments{2} \dodoubleempty}
\def\dodoubleargument\commando\[#1\][#2]\% 
  {\def\expectedarguments{2} \dodoubleempty}
\def\dosetupsomething\[#1\][#2]\% 
  {\def\expectedarguments{1} \dosetupsomething\empty}
```

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

\newif\iffirstargument
\newif\ifsecondargument
\newif\ifthirdargument
\newif\iffourthargument
\newif\ifthifthargument
\newif\ifsixthargument

The empty argument supplying macros mentioned before, look like:

\dotripleempty \command
\doquadrupleempty \command
\doquintupleempty \command
\dosixtupleempty \command

So \dodoubleempty leads to:

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

Depending of the generosity 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\dogetargument}
  \ifnum\expectedarguments>\noexpectedarguments
    \writestatus
    \{setup\}
    \{\expectedarguments\space argument(s) expected in line \the\inputlineno\space\}%
\let\expectedarguments=\noexpectedarguments
\dogetargument#1#2}}
143 \def\getsingleempty#1#2#3%
{\def\dodogetargument%
\dogetargument#1#2\firstargumenttrue\firstargumentfalse}
144 \def\getdoubleempty#1#2#3%
{\def\dodogetargument#1##1#2%
{\def\dodogetargument%
\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%
\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%
\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%
\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%
These maybe too mysterious macros enable us to handle more than one setup at once.

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

\begin{verbatim}
\def\dodefinesomething[#1][#2]{\getparameters[\??xx#1][#2]}
\def\definesomething{%
  \dodoubleargumentwithset\dodefinesomething}
\end{verbatim}

Which accepts calls like:

\texttt{\definesomething[alfa,beta,...][variable=...,...]}

Now a whole bunch of variables like \texttt{\@@xxalfavariable} and \texttt{\@@xxbetavariable} is defined.
Setups can be optional. A command expecting a setup is prefixed by \texttt{\textbackslash complex}, a command without one gets the prefix \texttt{\textbackslash simple}. Commands like this can be defined by:

\texttt{\textbackslash complexorsimple \{command\}}

When \texttt{\textbackslash command} is followed by a [setup], then

\texttt{\textbackslash complexcommand \{setup\}}

executes, else we get

\texttt{\textbackslash simplecommand}

An alternative for \texttt{\textbackslash complexorsimple} is:

\texttt{\textbackslash complexorsimpleempty \{command\}}

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

\texttt{\textbackslash complexcommando \{setup\}}
\texttt{\textbackslash complexcommando []}

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

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

\texttt{\textbackslash def\textbackslash complexorsimpleempty#1\%}
{\texttt{\textbackslash doifnextcharelse{[]}}}
{\texttt{\textbackslash firstargumenttrue\texttt{\textbackslash getvalue{\textbackslash s!complex#1}}} \texttt{\textbackslash firstargumentfalse\texttt{\textbackslash getvalue{\textbackslash s!complex#1}}}}
The previous commands are used that often that we found it worthwhile to offer two more alternatives.

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

160 \def\setnameofcommand#1#2% watch the \donottest
{\setnameofcommand{#2}%
\OE\def\OE#2\OE\{\OE\donottest\OE#1\OE\{\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.

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.
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 PPCTeX.

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

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

166 \def\dodoublegroupempty#1{% 
  \{\#1\ifx\next\bgroup \let\expectedarguments=\noexpectedarguments 
  \def\next{\#1{\#1}\dogetargument}\% 
  \else 
  \ifnum\expectedarguments>\noexpectedarguments 
  \writestatus{setup} 
  \{\expectedarguments\space argument(s) expected in line \the\inputlineno\space\} \% 
  \fi 
  \let\expectedarguments=\noexpectedarguments 
  \def\next{\#2\dogetargument{}}\% 
  \fi\fi\fi 
  \next\% 
  \futurelet\next\nextnext\} 

167 \def\dotriplegroupempty#1{% 
  \{\#1\ifx\next\bgroup \let\expectedarguments=\noexpectedarguments 
  \def\next{\#1{\#1{\#1}\dogetargument}}\% 
  \else 
  \ifnum\expectedarguments>\noexpectedarguments 
  \writestatus{setup} 
  \{\expectedarguments\space argument(s) expected in line \the\inputlineno\space\} \% 
  \fi 
  \let\expectedarguments=\noexpectedarguments 
  \def\next{\#2\dogetargument{}}\% 
  \fi\fi\fi 
  \next\% 
  \futurelet\next\nextnext\}
These macros explicitly 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 {a} {b} {c}
\dotriplegroupempty\test {a} {b}
\dotriplegroupempty\test {a}
\dotriplegroupempty\test
\{a\}
\{b\}

And alike.

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

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

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 width of the identification string with the macro \statuswidth.

\def\statuswidth{15}
\def\writestring{%\immediate\write16}
\def\writeline{%\writestring{}}
\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
\fi

\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).

\newif\ifdebuggerinfo
\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.

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

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

\protect
General

\!!box \ 4
\!!count \ 4
\!!depth \ 4
\!!dimen \ 4
\!!done \ 4
\!!height \ 4
\!!string \ 4
\!!toks \ 4
\!!width \ 4

?? \ 4

\@active \ 3
\@alignment \ 3
\@begingroup \ 3
\@comment \ 3
\@endgroup \ 3
\@endline \ 3
\@escape \ 3
\@ignore \ 3
\@letter \ 3
\@mathshift \ 3
\@other \ 3
\@parameter \ 3
\@space \ 3
\@subscript \ 3
\@superscript \ 3
\@EA \ 4

\abortinputifdefined \ 1

\@c \ 4
\@checkparameters \ 27
\@commalistelement \ 28
\@complexorsimple \ 33
\@complexorsimpleempty \ 33
\@copyparameters \ 27

\@debuggerinfo \ 37
\@definecomplexorsimple \ 34
\@definecomplexorsimpleempty \ 34
\@definestartstopcommand \ 34
\@doassign \ 25
\@doassigndefault \ 25
\@docopyvalue \ 24
\@dodoubleargument \ 29
\@dodoubleargumentwithset \ 32
\@dodoubleempty \ 30
\@dodoubleemptywithset \ 32
\@dodoubleemptygroupempty \ 35
\@dogetvalue \ 24
\@doif \ 10

\@doalldefinedelse \ 7
\@doifassignmentelse \ 27
\@doifcommon \ 13
\@doifcommonelse \ 13
\@doifdefined \ 7
\@doifdefinedelse \ 7
\@doifelse \ 10
\@doifempty \ 12
\@doifemptyelse \ 12
\@doifinset \ 12
\@doifinsetelse \ 12
\@doifnotinstringelse \ 21
\@doifnextcharelse \ 5
\@doifnot \ 10
\@doifnotcommon \ 13
\@doifnotempty \ 12
\@doifnotinset \ 12
\@doifnumberelse \ 22
\@doifundefined \ 7
\@doifundefinedelse \ 7
\@donottest \ 6, 10
\@doquadrupleargument \ 29
\@doquadrupleempty \ 30
\@doquintupleargument \ 29
\@doquintupleempty \ 30
\@dorisetvalue \ 24
\@dosetvalue \ 24
\@dosetvalue \ 24
\@dosinglegroupempty \ 35
\@dodoubleargument \ 29
\@dodoubleargument \ 29
\@dodoubleargumentwithset \ 32
\@dodoubleempty \ 30
\@dodoubleempty \ 30
\@dodoubleemptywithset \ 32
\@dodoubleemptygroupempty \ 35
\@dodoubleemptygroupempty \ 35

\@e! \ 4
\@expanded \ 4

\@firstcharacter \ 21
\@forgetparameters \ 26

\getcommacommandsize \ 28
\getcommalistsize \ 28
\getemptyparameters \ 27
\getparameters \ 26
\getfirstcharacter \ 21
\getfromcommacommand \ 28
\getfromcommalist \ 28
\getparameters \ 26
\getvalue 6
\gobble...arguments 4
\gobbleoneargument 4

\ifCONTEXT 3
\ifififthargument 30
\ififirstargument 30
\ififourthargument 30
\ififparameters 27
\ifisecondargument 30
\ifsixthargument 30
\ifthirdargument 30
\letvalue 6

\makerawcommalist 23
\normalspace 3

\p! 4
\processaction 17
\processallactionsinset 17
\processcommacommand 15
\processcommalist 15
\processcommalistwithparameters 15
\processfirstactioninset 17
\processunexpandedcommalist 24
\protect 2
\rawdoinsetelse 23

\rawprocessaction 23
\rawprocesscommalist 23
\resetvalue 6

\s! 4
\scratchbox 3
\scratchcounter 3
\scratchdimen 3
\scratchmuskip 3
\scratchskip 3
\scratchtoks ifdone 3
\setevalue 6
\setgvalue 6
\setvalue 6
\setvalue 6
\statuswidth 36

\undoassign 25
\unexpanded 6
\unexpandedprocessaction 20
\unexpandedprocessallactionsinset 20
\unexpandedprocessfirstactioninset 20
\unprotect 2

\v! 4

\wait 36
\writeline 36
\writestatus 36
\writestring 36