context

Taco Hoekwater

CONTEXT System Macros
part 1: General macros

keywords
macro programming, context, api, general

abstract
All large macro packages for TEX have the need for a number of low–level macros to easy the programming effort. This is definately true for the CONTEXT package, where the extensive use of key–value pairs and the multilingual interface introduce extra complications to the already tricky art of TEX programming.
Some of these internal macros are real gems, and nearly all of them can also be used in your own source documents. Although most of CONTEXT is described in the source code, sometimes the explanations are too technical for a casual user, and some of the documentation, especially the examples, is still in dutch.
This series of articles will try to highlight the most usable commands of the internal macro layer, using english examples, and removing most of the technical stuff (like the definitions of the macros themselves, and the optimization history).
Disclaimer: Quite a lot of the explanation text is copied from source code documentation by Hans Hagen. Always assume that the errors are mine and the good jokes are his.

Introduction

This article will mostly deal with the file syst-gen.tex. All except the first couple of macros appear in this file, which is input immediately after the inclusion of a stripped down version of the plain TEX format. Most of the following macros are very basic, most of them are related to programming constructs like flow control (`\if` statements), data structures (comma separated lists), and definitions.

Rudiments

`\contextversion` contains the CONTEXT version string. If you need to make sure you are running under CONTEXT, check for this macro. It is not defined in syst-gen but in `context.tex`. This is because syst-gen is sometimes loaded under LATEX.

`\contextversion`

The expansion of the macro is something like this: 2002.5.24

These macros delimit code that is executed conditionally. `\beginTEX` and `\beginETEX` are mutually exclusive, depending on whether or not the format file was compiled under an ε-TEX–enabled executable. A typical way of setting up your code to use ε-TEX where available is like this:

`\beginTEX`
`\endTEX`
`\beginETEX`
`\endETEX`
`\beginOMEGA`
`\endOMEGA`

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

\beginETEX \ifcsname
\def\ifundefined#1%
  {\unless\ifcsname#1\endcsname}
\endETEX
```

Code delimited by `\beginOMEGA` ... `\endOMEGA` is only executed if CONTEXT runs under OMEGA, and ignored otherwise.

The optional argument after the \begin... can be used to give information to the viewer: the example above will print the following string to the terminal:

```
system (E-TEX) : [line 833] \ifcsname
```

\abortinputifdefined

Because modules can be used in various contexts, we want to be able to prevent macor files from 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.

For example, syst-gen implements \writestatus, and therefore it starts with:

```
\abortinputifdefined\writestatus
```

Actually you don't need this macro for modules, since \usemodule does it's own bookkeeping. It is intended for files that are loaded via the TEX primitive \input.

\protect
\unprotect

We can shield macros from users by using some special characters in their names. Some characters that TEX normally does not consider to be letters (and therefore 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{alfa}
\protect
```

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

The protection/unprotection commands can be nested (unlike \makeatletter in LATEX). This nesting is a convenience, since it allows one to use the protection pair regardless of whether protection is already turned on.

When the nesting becomes deeper than one level, the system reports the current protection level.

It is a good habit to always start your macro files with \unprotect and end them with \protect.

Mnemonics and aliases

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

In CONTEXT we sometimes manipulate the ⟨*catcodes*⟩ of certain characters. Because we are not that good at numbers, we introduce some symbolic names. This makies it easier to key in things like this:

```
\catcode`@\@@letter
```

If you don't understand the names of the macros, you can look them up in the TEXbook.

Two of these symbolic names are actually used so often that they even have non-protected aliases: \other and \active.

\normalspace

We often need a space as defined in PLAIN TEX. Because CONTEXT cannot be sure that \space is not redefined, it internally uses an alias: \normalspace.

When dealing with CONTEXT, please remember to **never** (re-)define macros that start with \normal.... Weird, unexpected things can, and probably will, occur!

These are constant counters, corresponding to 0, 1 and −1.

\zeropoint
\plusone
\minusone

Temporary variables

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. The others are analogous scratch registers.

\scratchcounter
\scratchdimen
\scratchskip
\scratchmuskip
\scratchbox
\globalscratchbox
\scratchtoks
\ifdone
\!!

A warning is in order here. Yes, you can use these registers in your own macrocode. But **not** across calls to internal CONTEXT macros. While CONTEXT always makes sure that the registers are cleared on entry to the call, it usually doesn't bother to restore your user supplied value when it returns.

CONTEXT also uses a rather large collection of other internal scratch registers. Their names all look like this: `\!!XXXXXY`, where `XXXXX` is something like 'count' or 'depth', and `Y` is a letter starting from `a` (e.g. `\!!counta`). The fact that their names start with `!!` is a clear statement: Don't touch them, it's dangerous. If you need scratch registers, define your own.

CONTEXT uses yet another set of constants and variables to store all sorts of string values in (macro names occupy less space in TEX's memory than the strings themselves).

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

For this reason, you should also not touch the definitions of macros that start with `\s!` (constant string), `\c!` (constant), `\p!` (parameter), `\v!` (variable), `\@@` (multi–lingual interface parameter expansion), `\??` (multi–lingual interface parameter call).

Redefining these macros can have disastrous results.

Expansion control

When in unprotected mode, to be entered with `\unprotect`, one can use `\@EA` as equivalent of `\expandafter`. `\@EAEA` expands to two expandafters, `\@EAEAEA` to three, and the last one expands to `\@EA\@EAEAEA\@EA`.

\@EA
\@EAEA
\@EAEAEA
\@EAEAEAEAEAEA

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

\expanded

```
\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.

These two commands make macros more readable by hiding a lot of `\expandafter`'s. They expand the arguments after the first command.

\expandoneargafter
\expandtwoargsafter
\fullexpandoneargafter
\fullexpandtwoargsafter

```
\expandoneargafter \command{\abc}
\expandtwoargsafter\command{\abc}{\def}
\fullexpandoneargafter \command{\abc}
\fullexpandtwoargsafter\command{\abc}{\def}
```

These commands expect the arguments to be macros.

When expansion of a macro gives problems we can precede it by `\unexpanded`, like so:

\unexpanded

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

(if you are not familiar with CONTEXT: this is the same command as `\protect` in LATEX). It seems that protection is one of the burdens of developers of packages, so maybe that's why in ε-TEX protection is solved in a more robust way. CONTEXT uses that (more robust) solution if it is available, and otherwise tries it's best to emulate it using rather tricky macros.

Expansion problems can get quite complex. There are some other internal macros that can help harnassing it, but it is fairly unlikely that you will need them. If you believe you do, read the syst-gen source code.

Argument grabbing and handling

\gobbleoneargument
\gobbletwoarguments
\gobblethreearguments
\gobble...arguments

This set of macros does nothing, except that they get rid of a number of arguments, up to ten arguments altogether. This type of macro is especially useful when you don't really need the user supplied argument(s) to your macro.

For example, assume you need a macro that would be called like this:

```
\checkoddpage{this page is odd}
```

The 'best' definition for that macro is this:

```
\def\checkoddpage{%
  \ifodd\pageno
    \expandafter\message
  \else
    \expandafter\gobbleoneargument
  \fi
}
```

The 'simplistic' alternative macro definition:

```
\def\checkoddpage#1{%
  \ifodd\pageno
    \message{#1}%
  \fi
}
```

actually runs slower, since the argument is scanned twice: once by \checkoddpage, and once by \message.

\firstofoneargument
\firstoftwoarguments
\firstofthreearguments
\firstoffourarguments
\secondoftwoarguments
\secondofthreearguments
\secondoffourarguments
\thirdofthreearguments
\thirdoffourarguments
\fourthoffourarguments

These macros are trivial, but quite important in some applications. Here is the definition, which is also the example code:

```
\long\def\firstofoneargument          #1{#1}
\long\def\firstoftwoarguments       #1#2{#1}
\long\def\firstofthreearguments   #1#2#3{#1}
\long\def\firstoffourarguments  #1#2#3#4{#1}
\long\def\secondoftwoarguments      #1#2{#2}
\long\def\secondofthreearguments  #1#2#3{#2}
\long\def\secondoffourarguments #1#2#3#4{#2}
\long\def\thirdofthreearguments   #1#2#3{#3}
\long\def\thirdoffourarguments  #1#2#3#4{#3}
\long\def\fourthoffourarguments #1#2#3#4{#4}
```

The need for these commands appears when you have to strip braces from a (saved) argument. For instance, when you have a list with 'subelements', the list expansion tends to look like this:

```
{{0}{0},{0}{1},{1}{0},{1}{1}}
```

(A two by two matrix, stored as a commalist). Imagine that you want to grab the row numbers and do something with it. This definition:

```
\def\doprocessrow#1{...what i really want to do...}
\def\processrow#1{\expandafter\firstoftwoarguments\doprocessrow}
```

```
\processcommalist[{0}{0},{0}{1},{1}{0},{1}{1}]\processrow
```

does the trick.

Definitions and assignments

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` `\setgvalue` `\setevalue` `\setxvalue` `\letvalue` `\getvalue` `\resetvalue`

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

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

The next macro can be very useful when using `\csname` like in: `\strippedcsname`

```
\csname if\strippedcsname\something\endcsname
```

This expands to `\ifsomething`.

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` `\complexorsimpleempty`

```
\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 later in their existence.

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

These commands are called as:

```
\definecomplexorsimple\command
```

Of course, we must have available

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

Using this construction saves a few strings now and then.

\definestartstopcommand

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

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

This expands to something like:

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

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

```
\definestartstopcommand\somecommand\e!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}
```

Actually, this macro is never used in ConTeXt, but it used to be part of constructions like `\placeformula`.

Branches and decisions

\doifnextcharelse

When Pragma ADE started using TeX in the late eighties, their first experiences with programming concerned a simple shell around LaTeX. The commands probably used most at Pragma ADE are the itemizing ones. One of those initial shell commands took care of an optional argument, that enabled the specification of the item symbol to be sued. Without understanding anything they were able to locate a LaTeX macro that could be used to inspect the next character.

It is that 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 {char} {then ...} {else ...}
```

\doifundefined
\doifdefined
\doifundefinedelse
\doifdefinedelse
\doifalldefinedelse

The standard way of testing if a macro is defined is comparing its meaning with another undefined one, aptly named `\undefined`. To garantee correct working of this set of macros, `\undefined` may **never** be defined by a user!

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

\doif
\doifelse
\doifnot

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 ...}
```

These macros test string equality of the (expanded) first two arguments.

We complete our set of conditionals with:

\doifempty
\doifemptyelse
\doifnotempty

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

This time, the string is not expanded. Remember to expand it yourself where needed.

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

\doifinset
\doifnotinset
\doifinsetelse

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

The second argument is the comma separated set of strings. If the first string expands 'empty', it is considered to be **always** in the set. The comma separated set is not expanded.

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

\doifcommon
\doifnotcommon
\doifcommonelse

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

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

\doifinstringelse
\doifincsnameelse

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

The next alternative proved to be upto twice as fast on tasks like checking reserved words in pretty verbatim typesetting. This is mainly due to the fact that passing (expanded) strings is much slower that passing a macro.

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

Where `\doifinstringelse` does as much expansion as possible, the latter alternative does minimal (one level) expansion.

The next macro executes a command depending of the outcome of a test on numerals.

\doifnumberelse

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

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

The definition of this macro is extremly ugly, or extremely beautiful, depending on how you feel about TEX macro expansion. It is the first of only a few that will actually appear in this series of articles, I promise.

```
\long\def\doifnumberelse#1%
  {\ifcase0\ifcase1#1\or\or\or\or\or\or\or\or\or\else1\fi\space
     \expandafter\secondoftwoarguments
   \else
     \expandafter\firstoftwoarguments
   \fi}
```

Cases

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
\processfirstactioninset
\processallactionsinset

```
\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]
```

This macro is most often used in the key–value parser, like in this (simplified) example, where the user has said `width=small` or something similar:

```
\processaction
    [\TESTwidth] % should expand to 'small'
    [small=>\message{small was chosen},
     medium=>\message{medium was chosen}]
```

You 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 ...}]
```

If the first argument is empty, this macro scans the list for the keyword `default` and executes the related action, if present. If the first argument is not empty but also 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`.

`\getfirstcharacter` `\firstcharacter`

Sometimes an action needs to be undertaken that depends only on the first character of something (for instance, checking if some string represents a number or not). This macro get this character and puts it in `\firstcharacter`.

```
\getfirstcharacter {string}
```

Comma separated lists

`\processcommalist`

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

```
\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)}

1: \processcommalist [\hbox{$a,b,c,d,e,f$}] \dosomething \par
2: \processcommalist [{a,b,c,d,e,f}]        \dosomething \par
3: \processcommalist [{a,b,c},d,e,f]        \dosomething \par
4: \processcommalist [a,b,{c,d,e},f]        \dosomething \par
5: \processcommalist [a{b,c},d,e,f]         \dosomething \par
6: \processcommalist [{a,b}c,d,e,f]         \dosomething \par
7: \processcommalist []                     \dosomething \par
8: \processcommalist [{[}]                  \dosomething \par
```

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

And here is the result:

1: (a, b, c, d, e, f)
2: (a,b,c,d,e,f)
3: (a,b,c)(d)(e)(f)
4: (a)(b)(c,d,e)(f)
5: (ab,c)(d)(e)(f)
6: (a,bc)(d)(e)(f)

7:
8: ([)

Quitting a commalist halfway can be done by using `\quitcommalist` within the execution macro. `\quitcommalist`

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

```
\expandafter\processcommalist\expandafter[\list]\dosomething
```

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

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

where `\stringset` is a predefined set, like:

```
\def\first{zeroeth,first}
\def\second{last}

\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 its limititations.

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

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

where `\dosomething` looks like:

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

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 not a problem for `\rawprocesscommalist`), because this macro does not compare anything. Expandable macros are permitted as search string for `\rawdoifinsetelse`. `\rawdoinsetelse` `\rawprocesscommalist` `\rawprocessaction`

```
\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 gain).

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%.

It's possible to get an element from a commalist or a command representing a commalist. `\getfromcommalist` `\getfromcommacommand` `\commalistelement` `\getcommalistsize` `\getcommacommandsize`

```
\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`.

Because 0, 1 and 2 are often asked for, the macros are optimized on those numbers.

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*⟩).

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

`\dogetcommalistelement`
`\dogetcommacommandelement`

For low level (fast) purposes, we can also use the next alternative, which can handle up to 8 elements at most.

```
\dogetcommalistelement1\from a,b,c\to\commalistelement
```

It's ugly, but it is very fast indeed. Keep in mind that this version does **not** strip leading spaces from the list items.

Assignments and parameters

`\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]
```

These macros are a syntactic rewrite for the 'set', 'clear' and 'initialize' actions:

```
\def\labelvariable{value} % \doassign
\def\labelvariable{}      % \undoassign
\doifundefined{\labelvariable}
   {\def\labelvariable{value}} % \doassignempty
```

`\getparameters`
`\geteparameters`
`\getgparameters`
`\forgetparameters`

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

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

Again, the label identifies the category a variable belongs to. The second argument can be a comma separated list of assignments. Duplicates are allowed, later appearances overrule earlier ones.

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

is equivalent to

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

Some explanation of the inner workings of ConTeXt is in order here to make sure the source is understandable for readers, since the actual internal usage is a bit more complex than this.

In the pre–multi–lingual (simple) stadium ConTeXt took the next approach. With these definitions (that are mainly there to conserve TeX's string space):

```
\def\??demo {@@demo}
\def\!!width {width}
\def\!!height {height}
```

calling

```
\getparameters
  [\??demo]
  [\!!width=one,
   \!!height=2]
```

lead to:

```
\def\@@demowidth{one}
\def\@@demoheight{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.

The call will typically somewhat look like this:

```
\getparameters
  [\??demo]
  [\c!width=\v!one,
   \c!height=2]
```

In the dutch interface, this would (e.g.) expand into:

```
\def\@@demobreedte{een}
\def\@@demohoogte{2}
```

but in the english interface it would become:

```
\def\@@demowidth{one}
\def\@@demoheight{2}
```

CONTEXT continues to function, because it never uses the explicit expansion, anywhere. It always relies on the `\s!` and `\v!` definitions (remember I told you not to touch them? this is why.)

`\getemptyparameters`

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

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

`\copyparameters`

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]
  [\c!width,\c!height]
```

Leads to (english version):

```
\def\@@internalwidth  {\@@externalwidth}
\def\@@internalheight {\@@externalheight}
```

\doifassignmentelse

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

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

Although a combined solution is possible, we prefer a separation between argument keywords and parameter assignments. The next command takes care of propper handling of such multi–faced commands.

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

\checkparameters
\ifparameters

A slightly different approach is `\checkparameters`, which also checks on the presence of a `=`, just like the previous macro.

```
\checkparameters[argument]
```

The boolean `\ifparameters` can be used afterwards to verify that there were assignments in the supplied argument.

\dosingleargument
\dodoubleargument
\dotripleargument
\doquadrupleargument
\doquintupleargument
\dosixtupleargument
\doseventupleargument

When working with delimited arguments, spaces and lineendings can interfere. The next set of macros uses TeX' internal scanner for grabbing everything between arguments. Forgive me the funny names.

```
\dosingleargument\command    = \command[#1]
\dodoubleargument\command    = \command[#1][#2]
\dotripleargument\command    = \command[#1][#2][#3]
\doquadrupleargument\command = \command[#1][#2][#3][#4]
\doquintupleargument\command = \command[#1][#2][#3][#4][#5]
\dosixtupleargument\command  = \command[#1][#2][#3][#4][#5][#6]
\doseventupleargument\command= \command[#1][#2][#3][#4][#5][#6][#7]
```

These macros are used in the following way:

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

\def\command%
  {\dodoubleargument\docommand}
```

So `\dodoubleargument` leads to:

```
\docommand[#1][#2]
\docommand[#1][]
\docommand[][]
```

\iffirstargument
\ifsecondargument
\ifthirdargument
\iffourthargument
\iffifthargument
\ifsixthargument
\ifseventhargument

The macros above insure that the resulting call always has the correct number of bracket pairs, even if the user did not supply all of the options. In this case, a number of trailing bracket pairs are empty. A set of related `\if` boolemas is initialized to give you access to the number of user supplied parameters.

\dosingleargumentwithset
\dodoubleargumentwithset
\dodoubleemptywithset
\dotripleargumentwithset
\dotripleemptywithset

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

```
\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.

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     \ineedTHREEarguments
\doquadruplegroupempty  \ineedFOURarguments
\doquintuplegroupempty  \ineedFIVEarguments
```

\dosinglegroupempty
\dodoublegroupempty
\dotriplegroupempty
\doquadruplegroupempty
\doquintuplegroupempty

where `\ineedONEargument` takes one and the others two and three arguments, etcetera. These macros were first needed in PPCHTEX. At first glance, the functionality sounds trivial. But in fact, it is not. Consider this expanded input:

```
\def\test#1{\message{#1}}
\dosinglegroupempty\test {a}Text
\dosinglegroupempty\test Text
```

In the last line, `#1` will **not** print the letter `T`, as would be 'normal' TEX behaviour.

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

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

And alike.

Just as their `[]`, they also set the `\ifXXXXargument` switches.

User interaction

This macro hardly needs explanation. It stops TEX until you input something.

\wait

Macros 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
\writeline
\writestatus
\statuswidth

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

Messages are formatted: the category appears within a fixed number of columns, on it's own, followed by a colon. One can provide the maximum width of the identification string with the macro `\statuswidth`.

`\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).

```
\debuggerinfo {subcategory} {message}
```

This message will only be output if `\debuggerinfo` is true. (it usually isn't). If you need to calculate something to be used inside the `message`, embrace your calculating code with `\ifdebuggerinfo` ... `\fi`.

Index