

The documented source of Memoize, Advice and CollArgs

Memoize v1.2.0, Advice v1.1.0, CollArgs v1.1.0
March 15, 2024

Sašo Živanović

✉ saso.zivanovic@guest.arnes.si
📧 spj.ff.uni-lj.si/zivanovic
🐙 github.com/sasozivanovic

This file contains the documented source code of package [Memoize](#) and, somewhat unconventionally, its two independently distributed auxiliary packages [Advice](#) and [CollArgs](#).

The source code of the \TeX parts of the package resides in `memoize.edtx`, `advice.edtx` and `collargs.edtx`. These files are written in [EasyDTX](#), a format of my own invention which is almost like the DTX format but eliminates the need for all those pesky `macrocode` environments: Any line introduced by a single comment counts as documentation, and to top it off, documentation lines may be indented. An `.edtx` file is converted to a `.dtx` by a little Perl script called `edtx2dtx`; there is also a rudimentary Emacs mode, implemented in `easydoctex-mode.el`, which takes care of fontification, indentation, and forward and inverse search.

The `.edtx` files contain the code for all three formats supported by the three packages — \LaTeX (guard `latex`), plain \TeX (guard `plain`) and Con \TeX t (guard `context`) — but upon reading the code, it will quickly become clear that Memoize was first developed for \LaTeX . In §1, we manually define whatever \LaTeX tools are “missing” in plain \TeX and Con \TeX t. Even worse, Con \TeX t code is often just the same as plain \TeX code, even in cases where I’m sure Con \TeX t offers the relevant tools. This nicely proves that I have no clue about Con \TeX t. If you are willing to Con \TeX t-ualize my code — please do so, your help is welcome!

The runtimes of Memoize (and also Advice) comprise of more than just the main runtime for each format. Memoize ships with two additional stub packages, `nomemoize` and `memoizable`, and a \TeX -based extraction script `memoize-extract-one`; Advice optionally offers a TikZ support defined in `advice-tikz.code.tex`. For the relation between guards and runtimes, consult the core of the `.ins` files below.

memoize.ins

```
\generate{%
  \file{memoize.sty}{\from{memoize.dtx}{mmz,latex}}%
  \file{memoize.tex}{\from{memoize.dtx}{mmz,plain}}%
  \file{t-memoize.tex}{\from{memoize.dtx}{mmz,context}}%
  \file{nomemoize.sty}{\from{memoize.dtx}{nommmz,latex}}%
  \file{nomemoize.tex}{\from{memoize.dtx}{nommmz,plain}}%
  \file{t-nomemoize.tex}{\from{memoize.dtx}{nommmz,context}}%
  \file{memoizable.sty}{\from{memoize.dtx}{mmzable,latex}}%
  \file{memoizable.tex}{\from{memoize.dtx}{mmzable,plain}}%
  \file{t-memoizable.tex}{\from{memoize.dtx}{mmzable,context}}%
  \file{memoizable.code.tex}{\from{memoize.dtx}{mmzable,generic}}%
  \file{memoize-extract-one.tex}{\from{memoize.dtx}{extract-one}}%
  \file{memoize-biblatex.code.tex}{\from{memoize.dtx}{biblatex}}%
}
```

advice.ins

```
\file{advice.sty}{\from{advice.dtx}{main,latex}}%
\file{advice.tex}{\from{advice.dtx}{main,plain}}%
\file{t-advice.tex}{\from{advice.dtx}{main,context}}%
\file{advice-tikz.code.tex}{\from{advice.dtx}{tikz}}%
```

collargs.ins

```
\file{collargs.sty}{\from{collargs.dtx}{latex}}%
\file{collargs.tex}{\from{collargs.dtx}{plain}}%
\file{t-collargs.tex}{\from{collargs.dtx}{context}}%
```

Memoize also contains two scripts, `memoize-extract` and `memoize-clean`. Both come in two functionally equivalent implementations: Perl (`.pl`) and a Python (`.py`). Their code is listed in §9.

Contents

| | | |
|----------|---------------------------------------------------------------------|------------|
| 1 | First things first | 3 |
| 2 | The basic configuration | 7 |
| 3 | Memoization | 11 |
| 3.1 | Manual memoization | 11 |
| 3.2 | The memoization process | 15 |
| 3.3 | Context | 24 |
| 3.4 | C-memos | 25 |
| 3.5 | Cc-memos | 27 |
| 3.6 | The externs | 30 |
| 4 | Extraction | 38 |
| 4.1 | Extraction mode and method | 38 |
| 4.2 | The record files | 40 |
| 4.2.1 | The .mmz file | 41 |
| 4.2.2 | The shell scripts | 42 |
| 4.2.3 | The Makefile | 43 |
| 4.3 | T _E X-based extraction | 44 |
| 4.3.1 | memoize-extract-one.tex | 46 |
| 5 | Automemoization | 48 |
| 5.1 | L ^A T _E X-specific handlers | 52 |
| 6 | Support for various classes and packages | 54 |
| 6.1 | PGF | 54 |
| 6.2 | TikZ | 55 |
| 6.3 | Forest | 55 |
| 6.4 | Beamer | 55 |
| 6.5 | Biblatex | 56 |
| 7 | Initialization | 60 |
| 8 | Auxiliary packages | 63 |
| 8.1 | Extending commands and environments with Advice | 63 |
| 8.1.1 | Installation into a keypath | 63 |
| 8.1.2 | Submitting a command or environment | 66 |
| 8.1.3 | Executing a handled command | 72 |
| 8.1.4 | Environments | 74 |
| 8.1.5 | Error messages | 76 |
| 8.1.6 | Tracing | 77 |
| 8.1.7 | The TikZ collector | 79 |
| 8.2 | Argument collection with CollArgs | 80 |
| 8.2.1 | The keys | 82 |
| 8.2.2 | The central loop | 87 |
| 8.2.3 | Auxiliary macros | 89 |
| 8.2.4 | The handlers | 94 |
| 8.2.5 | The verbatim modes | 112 |
| 8.2.6 | Transition between the verbatim and the non-verbatim mode | 117 |
| 9 | The scripts | 127 |
| 9.1 | The Perl extraction script memoize-extract.pl | 127 |
| 9.2 | The Python extraction script memoize-extract.py | 139 |
| 9.3 | The Perl clean-up script memoize-clean.pl | 147 |
| 9.4 | The Python clean-up script memoize-clean.py | 150 |

1 First things first

Identification of memoize, memoizable and nomemoize.

```
1 <mmz>
2 <latex> \ProvidesPackage{memoize}[2024/03/15 v1.2.0 Fast and flexible externalization]
3 <context> %D \module[
4 <context> %D         file=t-memoize.tex,
5 <context> %D         version=1.2.0,
6 <context> %D         title=Memoize,
7 <context> %D         subtitle=Fast and flexible externalization,
8 <context> %D         author=Saso Zivanovic,
9 <context> %D         date=2024-03-15,
10 <context> %D         copyright=Saso Zivanovic,
11 <context> %D         license=LPPL,
12 <context> %D ]
13 <context> \writestatus{loading}{ConTeXt User Module / memoize}
14 <context> \unprotect
15 <context> \startmodule[memoize]
16 <plain> % Package memoize 2024/03/15 v1.2.0
17 </mmz>
18 <mmzable>
19 <latex> \ProvidesPackage{memoizable}[2024/03/15 v1.2.0 A programmer's stub for Memoize]
20 <context> %D \module[
21 <context> %D         file=t-memoizable.tex,
22 <context> %D         version=1.2.0,
23 <context> %D         title=Memoizable,
24 <context> %D         subtitle=A programmer's stub for Memoize,
25 <context> %D         author=Saso Zivanovic,
26 <context> %D         date=2024-03-15,
27 <context> %D         copyright=Saso Zivanovic,
28 <context> %D         license=LPPL,
29 <context> %D ]
30 <context> \writestatus{loading}{ConTeXt User Module / memoizable}
31 <context> \unprotect
32 <context> \startmodule[memoizable]
33 <plain> % Package memoizable 2024/03/15 v1.2.0
34 </mmzable>
35 <nommz>
36 <latex> \ProvidesPackage{nomemoize}[2024/03/15 v1.2.0 A no-op stub for Memoize]
37 <context> %D \module[
38 <context> %D         file=t-nomemoize.tex,
39 <context> %D         version=1.2.0,
40 <context> %D         title=Memoize,
41 <context> %D         subtitle=A no-op stub for Memoize,
42 <context> %D         author=Saso Zivanovic,
43 <context> %D         date=2024-03-15,
44 <context> %D         copyright=Saso Zivanovic,
45 <context> %D         license=LPPL,
46 <context> %D ]
47 <context> \writestatus{loading}{ConTeXt User Module / nomemoize}
48 <context> \unprotect
49 <context> \startmodule[nomemoize]
50 <mmz> % Package nomemoize 2024/03/15 v1.2.0
51 </nommz>
```

Required packages and L^AT_EXization of plain T_EX and ConT_EXt.

```
52 <*(mmz, mmzable, nommz) & (plain, context)>
53 \input miniltx
54 </(mmz, mmzable, nommz) & (plain, context)>
```

Some stuff which is “missing” in miniltx, copied here from latex.ltx.

```

55 <*mmz & (plain, context)>
56 \def\PackageWarning#1#2{%
57     \newlinechar``^^J\def\MessageBreak{^^J\space\space#1: }%
58     \message{#1: #2}}
59 </mmz & (plain, context)>

```

Same as the official definition, but without `\outer`. Needed for record file declarations.

```

60 <*mmz & plain>
61 \def\newtoks{\alloc@5\toks\toksdef\@cclvi}
62 \def\newwrite{\alloc@7\write\chardef\sixt@n}
63 </mmz & plain>

```

I can't really write any code without `etoolbox` ...

```

64 <*mmz>
65 <latex>\RequirePackage{etoolbox}
66 <plain, context>\input etoolbox-generic

```

Setup the `memoize` namespace in `LuaTeX`.

```

67 \ifdefined\luatexversion
68     \directlua{memoize = {}}
69 \fi

```

`pdfdoccmds.sty` eases access to some PDF primitives, but I cannot manage to load it in `ConTeXt`, even if it's supposed to be a generic package. So let's load `pdfdoccmds.lua` and copy-paste what we need from `pdfdoccmds.sty`.

```

70 <latex>\RequirePackage{pdfdoccmds}
71 <plain>\input pdfdoccmds.sty
72 <*context>
73 \directlua{%
74     require("pdfdoccmds")
75     tex.enableprimitives('pdf@', {'draftmode'})
76 }
77 \long\def\pdf@mdfivesum#1{%
78     \directlua{%
79         oberdiek.pdfdoccmds.mdfivesum("\luaescapestring{#1}", "byte")%
80     }%
81 }%
82 \def\pdf@system#1{%
83     \directlua{%
84         oberdiek.pdfdoccmds.system("\luaescapestring{#1}")%
85     }%
86 }
87 \let\pdf@primitive\primitive

```

Lua function `oberdiek.pdfdoccmds.filesize` requires the `kpse` library, which is not loaded in `ConTeXt`, see github.com/latex3/lua-uni-algos/issues/3, so we define our own `filesize` function.

```

88 \directlua{%
89     function memoize.filesize(filename)
90         local filehandle = io.open(filename, "r")

```

We can't easily use `~=`, as `~` is an active character, so the `else` workaround.

```

91         if filehandle == nil then
92             else
93                 tex.write(filehandle:seek("end"))
94                 io.close(filehandle)
95             end
96         end

```

```

97 }%
98 \def\pdf@filesize#1{%
99   \directlua{memoize.filesize("\luaescapestring{#1}")}%
100 }
101 </context>

```

Take care of some further differences between the engines.

```

102 \ifdef\pdftexversion{%
103 }{%
104   \def\pdfhorigin{1true in}%
105   \def\pdfvorigin{1true in}%
106   \ifdef\XeTeXversion{%
107     \let\quitvmode\leavevmode
108   }{%
109     \ifdef\luatexversion{%
110       \let\pdfpagewidth\pagewidth
111       \let\pdfpageheight\pageheight
112       \def\pdfmajorversion{\pdfvariable majorversion}%
113       \def\pdfminorversion{\pdfvariable minorversion}%
114     }{%
115       \PackageError{memoize}{Support for this TeX engine is not implemented}{}%
116     }%
117   }%
118 }
119 </mmz>

```

In ConT_EXt, `\unexpanded` means `\protected`, and the usual `\unexpanded` is available as `\normalunexpanded`. Option one: use dtx guards to produce the correct control sequence. I tried this option. I find it ugly, and I keep forgetting to guard. Option two: `\let` an internal control sequence, like `\mmz@unexpanded`, to the correct thing, and use that all the time. I never tried this, but I find it ugly, too, and I guess I would forget to use the new control sequence, anyway. Option three: use `\unexpanded` in the `.dtx`, and `sed` through the generated ConT_EXt files to replace all its occurrences by `\normalunexpanded`. Oh yeah!

Load `pgfkeys` in `nomemoize` and `memoizable`. Not necessary in `memoize`, as it is already loaded by `CollArgs`.

```

120 <*nommz, mmzable>
121 <latex>\RequirePackage{pgfkeys}
122 <plain>\input pgfkeys
123 <context>\input t-pgfkey
124 </nommz, mmzable>

```

Different formats of `memoizable` merely load `memoizable.code.tex`, which exists so that `memoizable` can be easily loaded by generic code, like a `tikz` library.

```

125 <mmzable&!generic>\input memoizable.code.tex

```

Shipout We will next load our own auxiliary package, `CollArgs`, but before we do that, we need to grab `\shipout` in plain T_EX. The problem is, `Memoize` needs to hack into the `shipout` routine, but it has best chances of working as intended if it redefines the *primitive* `\shipout`. However, `CollArgs` loads `pgfkeys`, which in turn (and perhaps with no for reason) loads `atbegshi`, which redefines `\shipout`. For details, see section 3.6. Below, we first check that the current meaning of `\shipout` is primitive, and then redefine it.

```

126 <*mmz>
127 <*plain>
128 \def\mmz@regular@shipout{%
129   \global\advance\mmzRegularPages1\relax
130   \mmz@primitive@shipout
131 }
132 \edef\mmz@temp{\string\shipout}%

```

```

133 \edef\mmz@tempa{\meaning\shipout}%
134 \ifx\mmz@temp\mmz@tempa
135   \let\mmz@primitive\shipout\shipout
136   \let\shipout\mmz@regular\shipout
137 \else
138   \PackageError{memoize}{Cannot grab \string\shipout, it is already redefined}{}%
139 \fi
140 </plain>

```

Our auxiliary package (^M§5.6.3, §8.2). We also need it in `nomemoize`, to collect manual environments.

```

141 <latex>\RequirePackage{advice}
142 <plain>\input advice
143 <context>\input t-advice
144 </mmz>

```

Loading order `memoize` and `nomemoize` are mutually exclusive, and `memoizable` must be loaded before either of them. `\mmz@loadstatus`: 1 = `memoize`, 2 = `memoizable`, 3 = `nomemoize`.

```

145 <*mmz,nommz>
146 \def\ifmmz@loadstatus#1{%
147   \ifnum#1=0\csname mmz@loadstatus\endcsname\relax
148     \expandafter\@firstoftwo
149   \else
150     \expandafter\@secondoftwo
151   \fi
152 }
153 </mmz,nommz>
154 <*mmz>
155 \ifmmz@loadstatus{3}{%
156   \PackageError{memoize}{Cannot load the package, as "nomemoize" is already
157     loaded. Memoization will NOT be in effect}{Packages "memoize" and
158     "nomemoize" are mutually exclusive, please load either one or the other.}%
159 <latex> \pgfkeys{/memoize/package options/.unknown/.code={}}
160 <latex> \ProcessPgfPackageOptions{/memoize/package options}
161   \endinput
162 }{}%
163 \ifmmz@loadstatus{2}{%
164   \PackageError{memoize}{Cannot load the package, as "memoizable" is already
165     loaded}{Package "memoizable" is loaded by packages which support
166     memoization. Memoize must be loaded before all such packages. The
167     compilation log can help you figure out which package loaded "memoizable";
168     please move
169 <latex>   "\string\usepackage{memoize}"
170 <plain>   "\string\input memoize"
171 <context> "\string\usemodule[memoize]"
172     before the
173 <latex>   "\string\usepackage"
174 <plain>   "\string\input"
175 <context> "\string\usemodule"
176     of that package.}%
177 <latex> \pgfkeys{/memoize/package options/.unknown/.code={}}
178 <latex> \ProcessPgfPackageOptions{/memoize/package options}
179   \endinput
180 }{}%
181 \ifmmz@loadstatus{1}{\endinput}{}%
182 \def\mmz@loadstatus{1}%
183 </mmz>
184 <*mmzable & generic>
185 \ifcsname mmz@loadstatus\endcsname\endinput\fi
186 \def\mmz@loadstatus{2}%
187 </mmzable & generic>

```

```

188 <*/nommz>
189 \ifmmz@loadstatus{1}{%
190   \PackageError{nomemoize}{Cannot load the package, as "memoize" is already
191     loaded; memoization will remain in effect}{Packages "memoize" and
192     "nomemoize" are mutually exclusive, please load either one or the other.}%
193   \endinput }{}%
194 \ifmmz@loadstatus{2}{%
195   \PackageError{nomemoize}{Cannot load the package, as "memoizable" is already
196     loaded}{Package "memoizable" is loaded by packages which support
197     memoization. (No)Memoize must be loaded before all such packages. The
198     compilation log can help you figure out which package loaded
199     "memoizable"; please move
200 <latex>   "\string\usepackage{nomemoize}"
201 <plain>   "\string\input memoize"
202 <context> "\string\usemodule[memoize]"
203     before the
204 <latex>   "\string\usepackage"
205 <plain>   "\string\input"
206 <context> "\string\usemodule"
207     of that package.}%
208   \endinput
209 }{}%
210 \ifmmz@loadstatus{3}{\endinput}{}%
211 \def\mmz@loadstatus{3}%
212 </nommz>

213 <*/mmz>

```

`\filetotoks` Read T_EX file #2 into token register #1 (under the current category code regime); `\toksapp` is defined in CollArgs.

```

214 \def\filetotoks#1#2{%
215   \immediate\openin0{#2}%
216   #1={}%
217   \loop
218   \unless\ifeof0
219     \read0 to \totoks@temp

```

We need the `\expandafters` for our `\toksapp` macro.

```

220   \expandafter\toksapp\expandafter#1\expandafter{\totoks@temp}%
221   \repeat
222   \immediate\closein0
223 }

```

Other little things.

```

224 \newif\ifmmz@temp
225 \newtoks\mmz@temptoks
226 \newbox\mmz@box
227 \newwrite\mmz@out

```

2 The basic configuration

`\mmzset` The user primarily interacts with Memoize through the pgfkeys-based configuration macro `\mmzset`, which executes keys in path `/mmz`. In `nomemoize` and `memoizable`, it exists as a no-op.

```

228 \def\mmzset#1{\pgfkeys{/mmz}{#1}\ignorespaces}
229 </mmz>
230 <*/nommz, mmzable & generic>
231 \def\mmzset#1{\ignorespaces}
232 </nommz, mmzable & generic>

```

\nommzkeys Any /mmz keys used outside of \mmzset must be declared by this macro for nomemoize package to work.

```
233 <mmz>\def\nommzkeys#1{}
234 <*nommz, mmzable & generic>
235 \def\nommzkeys{\pgfqkeys{/mmz}}
236 \pgfqkeys{/mmz}{.unknown/.code={\pgfkeysdef{\pgfkeyscurrentkey}{}}}
237 </nommz, mmzable & generic>
```

enable These keys set T_EX-style conditional \ifmemoize, used as the central on/off switch for the functionality of the package — it is inspected in \Memoize and by run conditions of automemoization

\ifmemoize handlers.

If used in the preamble, the effect of these keys is delayed until the beginning of the document. The delay is implemented through a special style, **begindocument**, which is executed at **begindocument** hook in L^AT_EX; in other formats, the user must invoke it manually (^M§5.1).

Nomemoize does not need the keys themselves, but it does need the underlying conditional — which will be always false.

```
238 <*mmz, nommz, mmzable & generic>
239 \newif\ifmemoize
240 </mmz, nommz, mmzable & generic>
241 <*mmz>
242 \mmzset{%
243   enable/.style={begindocument/.append code=\memoizetrue},
244   disable/.style={begindocument/.append code=\memoizefalse},
245   begindocument/.append style={
246     enable/.code=\memoizetrue,
247     disable/.code=\memoizefalse,
248   },
```

Memoize is enabled at the beginning of the document, unless explicitly disabled by the user in the preamble.

```
249   enable,
```

options Execute the given value as a keylist of Memoize settings.

```
250   options/.style={#1},
251 }
```

normal When Memoize is enabled, it can be in one of three modes (^M§2.4): normal, readonly, and **readonly** recompile. The numeric constants are defined below. The mode is stored in \mmz@mode, and only **recompile** matters in \Memoize (and \mmz@process@ccmemo).¹

```
252 \def\mmz@mode@normal{0}
253 \def\mmz@mode@readonly{1}
254 \def\mmz@mode@recompile{2}
255 \let\mmz@mode\mmz@mode@normal
256 \mmzset{%
257   normal/.code={\let\mmz@mode\mmz@mode@normal},
258   readonly/.code={\let\mmz@mode\mmz@mode@readonly},
259   recompile/.code={\let\mmz@mode\mmz@mode@recompile},
260 }
```

prefix Key **path** executes the given keylist in path /mmz/path, to determine the full *path prefix* to memo and extern files (^M§2.5,4.2): **relative**, true by default, determines whether the location of these files is relative to the current directory; **dir** sets their directory; and **prefix** sets the first, fixed part of their basename; the second part containing the MD5 sum(s) is not under user control, and neither is the suffix. These subkeys will be initialized a bit later, via **no memo dir**.

¹In fact, this code treats anything but 1 and 2 as normal.


```

261 \mmzset{%
262   prefix/.code={\mmz@parse@prefix{#1}},
263 }

```

\mmz@split@prefix This macro stores the detokenized expansion of #1 into `\mmz@prefix`, which it then splits into `\mmz@prefix@dir` and `\mmz@prefix@name` at the final /. The slash goes into `\mmz@prefix@dir`. If there is no slash, `\mmz@prefix@dir` is empty.

```

264 \begingroup
265 \catcode`\/=12
266 \gdef\mmz@parse@prefix#1{%
267   \edef\mmz@prefix{\detokenize\expandafter\expanded{#1}}}%
268   \def\mmz@prefix@dir{%
269     \def\mmz@prefix@name{%
270       \expandafter\mmz@parse@prefix@i\mmz@prefix/\mmz@eov
271     }
272   \gdef\mmz@parse@prefix@i#1/#2{%
273     \ifx\mmzeov#2%
274       \def\mmz@prefix@name{#1}%
275     \else
276       \appto\mmz@prefix@dir{#1/}%
277       \expandafter\mmz@parse@prefix@i\expandafter#2%
278     \fi
279   }
280 \endgroup

```

Key `prefix` concludes by performing two actions: it creates the given directory if `mkdir` is in effect, and notes the new prefix in record files (by eventually executing `record/prefix`, which typically puts a `\mmzPrefix` line in the `.mmz` file). In the preamble, only the final setting of `prefix` matters, so this key is only equipped with the action-triggering code at the beginning of the document.

```

281 \mmzset{%
282   begindocument/.append style={
283     prefix/.append code=\mmz@maybe@mkmemodir\mmz@record@prefix,
284   },

```

Consequently, the post-prefix-setting actions must be triggered manually at the beginning of the document. Below, we trigger directory creation; `record/prefix` will be called from `record/begin`, which is executed at the beginning of the document, so it shouldn't be mentioned here.

```

285   begindocument/.append code=\mmz@maybe@mkmemodir,
286 }

```

mkdir Should we create the memo/extern directory if it doesn't exist? And which command should we **mkdir command** use to create it? There is no initial value for the latter, because `mkdir` cannot be executed out of the box, but note that `extract=perl` and `extract=python` will set the extraction script with option `--mkdir` as the value of `mkdir command`.

```

287 \mmzset{
288   mkdir/.is if=\mmz@mkdir,
289   mkdir command/.store in=\mmz@mkdir@command,
290   mkdir command={},
291 }

```

The underlying conditional `\ifmmz@mkdir` is only ever used in `\mmz@maybe@mkmemodir` below, which is itself only executed at the end of `prefix` and in `begindocument`.

```

292 \newif\ifmmz@mkdir
293 \mmz@mkdirtrue

```

We only attempt to create the memo directory if `\ifmmz@mkdir` is in effect and if both `\mmz@mkdir@command` and `\mmz@prefix@dir` are specified (i.e. non-empty).

```

294 \def\mmz@maybe@mkmemodir{%
295   \ifmmz@mkdir
296     \ifdefempty\mmz@mkdir@command{}\{%
297       \ifdefempty\mmz@prefix@dir{}\{%
298         \mmz@remove@quotes{\mmz@prefix@dir}\mmz@temp
299         \pdf@system{\mmz@mkdir@command\space"\mmz@temp"}%
300       }%
301     }%
302   \fi
303 }

```

`memo dir` Shortcuts for two common settings of `path` keys. The default `no memo dir` will place the `no memo dir` memos and externs in the current directory, prefixed with `#1.`, where `#1` defaults to (unquoted) `\jobname`. Key `memo dir` places the memos and externs in a dedicated directory, `#1.memo.dir`; the filenames themselves have no prefix. Furthermore, `memo dir` triggers the creation of the directory.

```

304 \mmzset{%
305   memo dir/.style={prefix={#1.memo.dir/}},
306   memo dir/.default=\jobname,
307   no memo dir/.style={prefix={#1.}},
308   no memo dir/.default=\jobname,
309   no memo dir,
310 }

```

`\mmz@remove@quotes` This macro removes fully expands `#1`, detokenizes the expansion and then removes all double quotes the string. The result is stored in the control sequence given in `#2`.

We use this macro when we are passing a filename constructed from `\jobname` to external programs.

```

311 \def\mmz@remove@quotes#1#2{%
312   \def\mmz@remove@quotes@end{\let#2\mmz@temp}%
313   \def\mmz@temp{}%
314   \expanded{%
315     \noexpand\mmz@remove@quotes@i
316     \detokenize\expandafter{\expanded{#1}}%
317     "\noexpand\mmz@eov
318   }%
319 }
320 \def\mmz@remove@quotes@i{%
321   \CollectArgumentsRaw
322   {\collargsReturnPlain
323     \collargsNoDelimiterstrue
324     \collargsAppendExpandablePostprocessor{\the\collargsArg}}%
325   }%
326   {u"u\mmz@eov}%
327   \mmz@remove@quotes@ii
328 }
329 \def\mmz@remove@quotes@ii#1#2{%
330   \appto\mmz@temp{#1}%
331   \ifx&#2&%
332     \mmz@remove@quotes@end
333     \expandafter\@gobble
334   \else
335     \expandafter\@firstofone
336   \fi
337   {\mmz@remove@quotes@i#2\mmz@eov}%
338 }

```

ignore spaces The underlying conditional will be inspected by automemoization handlers, to maybe put `\ignorespaces` after the invocation of the handler.

```
339 \newif\ifmmz@ignorespaces
340 \mmzset{
341   ignore spaces/.is if=mmz@ignorespaces,
342 }
```

verbatim These keys are tricky. For one, there's **verbatim**, which sets all characters' category codes to other, and there's **verb**, which leaves braces untouched (well, honestly, it redefines them). But **no verbatim** Memoize itself doesn't really care about this detail — it only uses the underlying conditional `\ifmmz@verbatim`. It is `CollArgs` which cares about the difference between the “long” and the “short” verbatim, so we need to tell it about it. That's why the verbatim options “append themselves” to `\mmzRawCollectorOptions`, which is later passed on to `\CollectArgumentsRaw` as a part of its optional argument.

```
343 \newif\ifmmz@verbatim
344 \def\mmzRawCollectorOptions{}
345 \mmzset{
346   verbatim/.code={%
347     \def\mmzRawCollectorOptions{\collargsVerbatim}%
348     \mmz@verbatimtrue
349   },
350   verb/.code={%
351     \def\mmzRawCollectorOptions{\collargsVerb}%
352     \mmz@verbatimtrue
353   },
354   no verbatim/.code={%
355     \def\mmzRawCollectorOptions{\collargsNoVerbatim}%
356     \mmz@verbatimfalse
357   },
358 }
```

3 Memoization

3.1 Manual memoization

\mmz The core of this macro will be a simple invocation of `\Memoize`, but to get there, we have to collect the optional argument carefully, because we might have to collect the memoized code verbatim.

```
359 \protected\def\mmz{\futurelet\mmz@temp\mmz@i}
360 \def\mmz@i{%
```

Anyone who wants to call `\Memoize` must open a group, because `\Memoize` will close a group.

```
361   \begingroup
```

As the optional argument occurs after a control sequence (`\mmz`), any spaces were consumed and we can immediately test for the opening bracket.

```
362   \ifx\mmz@temp[%
363     \def\mmz@verbatim@fix{}}%
364     \expandafter\mmz@ii
365   \else
```

If there was no optional argument, the opening brace (or the unlikely single token) of our mandatory argument is already tokenized. If we are requested to memoize in a verbatim mode, this non-verbatim tokenization was wrong, so we will use option `\collargsFixFromNoVerbatim` to ask `CollArgs` to fix the situation. (`\mmz@verbatim@fix` will only be used in the verbatim mode.)

```
366     \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
```

No optional argument, so we can skip `\mmz@ii`.

```
367 \expandafter\mmz@iii
368 \fi
369 }
370 \def\mmz@ii[#1]{%
```

Apply the options given in the optional argument.

```
371 \mmzset{#1}%
372 \mmz@iii
373 }
374 \def\mmz@iii{%
```

In the non-verbatim mode, we avoid collecting the single mandatory argument using `\CollectArguments`.

```
375 \ifmmz@verbatim
376 \expandafter\mmz@do@verbatim
377 \else
378 \expandafter\mmz@do
379 \fi
380 }
```

This macro grabs the mandatory argument of `\mmz` and calls `\Memoize`.

```
381 \long\def\mmz@do#1{%
382 \Memoize{#1}{#1}%
383 }%
```

The following macro uses `\CollectArgumentsRaw` of package `CollArgs` (§8.2) to grab the argument verbatim; the appropriate verbatim mode triggering raw option was put in `\mmzRawCollectorOptions` by key `verb(atim)`. The macro also `\mmz@verbatim@fix` contains the potential request for a category code fix (§8.2.6).

```
384 \def\mmz@do@verbatim#1{%
385 \expanded{%
386 \noexpand\CollectArgumentsRaw{%
387 \noexpand\collargsCaller{\noexpand\mmz}%
388 \expandonce\mmzRawCollectorOptions
389 \mmz@verbatim@fix
390 }%
391 }{+m}\mmz@do
392 }
```

memoize (*env.*) The definition of the manual memoization environment proceeds along the same lines as the definition of `\mmz`, except that we also have to implement space-trimming, and that we will collect the environment using `\CollectArguments` in both the verbatim and the non-verbatim and mode.

We define the \LaTeX , plain \TeX and \ConTeXt environments in parallel. The definition of the plain \TeX and \ConTeXt version is complicated by the fact that space-trimming is affected by the presence vs. absence of the optional argument (for purposes of space-trimming, it counts as present even if it is empty).

```
393 <*\latex>
```

We define the \LaTeX environment using `\newenvironment`, which kindly grabs any spaces in front of the optional argument, if it exists — and if doesn't, we want to trim spaces at the beginning of the environment body anyway.

```
394 \newenvironment{memoize}[1][\mmz@noarg]{%
```

We close the environment right away. We'll collect the environment body, complete with the end-tag, so we have to reintroduce the end-tag somewhere. Another place would be after the invocation of `\Memoize`, but that would put memoization into a double group and `\mmzAfterMemoization` would not work.

```
395 \end{memoize}%
```

We open the group which will be closed by `\Memoize`.

```
396 \begin{group}
```

As with `\mmz` above, if there was no optional argument, we have to ask Collargs for a fix. The difference is that, as we have collected the optional argument via `\newcommand`, we have to test for its presence in a roundabout way.

```
397 \def\mmz@temp{#1}%
398 \ifx\mmz@temp\mmz@noarg
399 \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
400 \else
401 \def\mmz@verbatim@fix{}%
402 \mmzset{#1}%
403 \fi
404 \mmz@env@iii
405 }{}
406 \def\mmz@noarg{\mmz@noarg}
407 \end{group}
408 \plain\def\memoize{%
409 \context\def\startmemoize{%
410 \plain, context
411 \begin{group}
```

In plain `TeX` and `ConTeXt`, we don't have to worry about any spaces in front of the optional argument, as the environments are opened by a control sequence.

```
412 \futurelet\mmz@temp\mmz@env@i
413 }
414 \def\mmz@env@i{%
415 \ifx\mmz@temp[%
416 \def\mmz@verbatim@fix{}%
417 \expandafter\mmz@env@ii
418 \else
419 \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
420 \expandafter\mmz@env@iii
421 \fi
422 }
423 \def\mmz@env@ii[#1]{%
424 \mmzset{#1}%
425 \mmz@env@iii
426 }
427 \end{group}
428 \def\mmz@env@iii{%
429 \long\edef\mmz@do##1{%
```

`\unskip` will “trim” spaces at the end of the environment body.

```
430 \noexpand\Memoize{##1}{##1\unskip}%
431 }%
432 \expanded{%
433 \noexpand\CollectArgumentsRaw{%
```

`\CollectArgumentsRaw` will adapt the caller to the format automatically.

```
434 \noexpand\collargsCaller{memoize}%
```

verb(atim) is in here if it was requested.

```
435 \expandonce\mmzRawCollectorOptions
```

The category code fix, if needed.

```
436 \ifmmz@verbatim\mmz@verbatim@fix\fi
437 }%
```

Spaces at the beginning of the environment body are trimmed by setting the first argument to !t<space> and disappearing it with \collargsAppendExpandablePostprocessor{}; note that this removes any number of space tokens. \CollectArgumentsRaw automatically adapts the argument type b to the format.

```
438 }{&&\collargsAppendExpandablePostprocessor{}}!t{ }+b{memoize}}{\mmz@do}%
439 }%
440 </mmz>
```

\nommz We throw away the optional argument if present, and replace the opening brace with begin-group plus \memoizefalse. This way, the “argument” of \nommz will be processed in a group (with Memoize disabled) and even the verbatim code will work because the “argument” will not have been tokenized.

As a user command, \nommz has to make it into package nomemoize as well, and we’ll \let \mmz equal it there; it is not needed in mmzable.

```
441 <*mmz, nommz>
442 \protected\def\nommz#1#{%
443 \afterassignment\nommz@i
444 \let\mmz@temp
445 }
446 \def\nommz@i{%
447 \bgroup
448 \memoizefalse
449 }
450 <nommz>\let\mmz\nommz
```

nomemoize (env.) We throw away the optional argument and take care of the spaces at the beginning and at the end of the body.

```
451 <*latex>
452 \newenvironment{nomemoize}[1][ ]{%
453 \memoizefalse
454 \ignorespaces
455 }{%
456 \unskip
457 }
458 </latex>
459 <*plain, context>
460 <plain>\def\nomemoize{%
461 <context>\def\startnomemoize{%
```

Start a group to delimit \memoizefalse.

```
462 \begingroup
463 \memoizefalse
464 \futurelet\mmz@temp\nommz@env@i
465 }
466 \def\nommz@env@i{%
467 \ifx\mmz@temp[%]
468 \expandafter\nommz@env@ii
```

No optional argument, no problems with spaces.

```

469 \fi
470 }
471 \def\nommz@env@ii[#1]{%
472 \ignorespaces
473 }
474 <plain>\def\endmemoize{%
475 <context>\def\stopmemoize{%
476 \endgroup
477 \unskip
478 }
479 </plain,context>
480 <*nommz>
481 <plain,latex>\let\memoize\nomemoize
482 <plain,latex>\let\endmemoize\endnomemoize
483 <context>\let\startmemoize\startnomemoize
484 <context>\let\stopmemoize\stopnomemoize
485 </nommz>
486 </mmz,nommz>

```

3.2 The memoization process

\ifmemoizing This conditional is set to true when we start memoization (but not when we start regular compilation or utilization); it should never be set anywhere else. It is checked by **\Memoize** to prevent nested memoizations, deployed in advice run conditions set by **run only if memoizing**, etc.

```

487 <*mmz,nommz,mmzable & generic>
488 \newif\ifmemoizing

```

\ifinmemoize This conditional is set to true when we start either memoization or regular compilation (but not when we start utilization); it should never be set anywhere else. It is deployed in the default advice run conditions, making sure that automemoized commands are not handled even when we’re regularly compiling some code submitted to memoization.

```

489 \newif\ifinmemoize

```

\mmz@maybe@scantokens An auxiliary macro which rescans the given code using **\scantokens** if the verbatim mode is active. We also need it in **NoMemoize**, to properly grab verbatim manually memoized code.

```

490 </mmz,nommz,mmzable & generic>
491 <*mmz>
492 \def\mmz@maybe@scantokens{%
493 \ifmmz@verbatim
494 \expandafter\mmz@scantokens
495 \else
496 \expandafter\@firstofone
497 \fi
498 }

```

Without **\newlinechar=13**, **\scantokens** would see receive the entire argument as one long line — but it would not *see* the entire argument, but only up to the first newline character, effectively losing most of the tokens. (We need to manually save and restore **\newlinechar** because we don’t want to execute the memoized code in yet another group.)

```

499 \long\def\mmz@scantokens#1{%
500 \expanded{%
501 \newlinechar=13
502 \unexpanded{\scantokens{#1\endinput}}}%
503 \newlinechar=\the\newlinechar
504 }%
505 }

```

`\Memoize` Memoization is invoked by executing `\Memoize`. This macro is a decision hub. It test for the existence of the memos and externs associated with the memoized code, and takes the appropriate action (memoization: `\mmz@memoize`; regular compilation: `\mmz@compile`, utilization: `\mmz@process@cmemo` plus `\mmz@process@ccmemo` plus further complications) depending on the memoization mode (normal, readonly, recompile). Note that one should open a \TeX group prior to executing `\Memoize`, because `\Memoize` will close a group (^M§4.1).

`\Memoize` takes two arguments, which contain two potentially different versions of the code submitted to memoization: `#1` contains the code which $\langle code MD5 sum \rangle$ is computed off of, while `#2` contains the code which is actually executed during memoization and regular compilation. The arguments will contain the same code in the case of manual memoization, but they will differ in the case of automemoization, where the executable code will typically prefixed by `\AdviceOriginal`. As the two codes will be used not only by `\Memoize` but also by macros called from `\Memoize`, `\Memoize` stores them into dedicated toks registers, declared below.

```
506 \newtoks\mmz@mdfive@source
507 \newtoks\mmz@exec@source
```

Finally, the definition of the macro. In package NoMemoize, we should simply execute the code in the second argument. But in Memoize, we have work to do.

```
508 \let\Memoize\@secondoftwo
509 \long\def\Memoize#1#2{%
```

We store the first argument into token register `\mmz@mdfive@source` because we might have to include it in tracing info (when `trace` is in effect), or paste it into the c-memo (depending on include source in `cmemo`).

```
510 \mmz@mdfive@source{#1}%
```

We store the executable code in `\mmz@exec@source`. In the verbatim mode, the code will have to be rescanned. This is implemented by `\mmz@maybe@scantokens`, and we wrap the code into this macro right away, once and for all. Even more, we pre-expand `\mmz@maybe@scantokens` (three times), effectively applying the current `\ifmmz@verbatim` and eliminating the need to save and restore this conditional in `\mmz@compile`, which (regularly) compiles the code *after* closing the `\Memoize` group — after this pre-expansion, `\mmz@exec@source` will contain either `\mmz@scantokens{...}` or `\@firstofone{...}`.

```
511 \expandafter\expandafter\expandafter\expandafter
512 \expandafter\expandafter\expandafter
513 \mmz@exec@source
514 \expandafter\expandafter\expandafter\expandafter
515 \expandafter\expandafter\expandafter
516 {%
517 \mmz@maybe@scantokens{#2}%
518 }%
519 \mmz@trace@Memoize
```

In most branches below, we end up with regular compilation, so let this be the default action.

```
520 \let\mmz@action\mmz@compile
```

If Memoize is disabled, or if memoization is currently taking place, we will perform a regular compilation.

```
521 \ifmemoizing
522 \else
523 \ifmemoize
```

Compute $\langle code md5sum \rangle$ off of the first argument, and globally store it into `\mmz@code@mdfivesum` — globally, because we need it in utilization to include externs, but the `\Memoize` group is closed (by `\mmzMemo`) while inputting the cc-memo.

```
524 \xdef\mmz@code@mdfivesum{\pdf@mdfivesum{\the\mmz@mdfive@source}}%
525 \mmz@trace@code@mdfive
```


Recompile mode forces memoization.

```

526     \ifnum\mmz@mode=\mmz@mode@recompile\relax
527     \ifnum\pdf@draftmode=0
528     \let\mmz@action\mmz@memoize
529     \fi
530 \else

```

In the normal and the readonly mode, we try to utilize the memos. The c-memo comes first. If the c-memo does not exist (or if something is wrong with it), `\mmz@process@cmemo` (defined in §3.4) will set `\ifmmz@abort` to true. It might also set `\ifmmzUnmemoizable` which means we should compile normally regardless of the mode.

```

531     \mmz@process@cmemo
532     \ifmmzUnmemoizable
533     \mmz@trace@cmemo@unmemoizable
534 \else
535     \ifmmz@abort

```

If there is no c-memo, or it is invalid, we memoize, unless the read-only mode is in effect.

```

536     \mmz@trace@process@cmemo@fail
537     \ifnum\mmz@mode=\mmz@mode@readonly\relax
538     \else
539     \ifnum\pdf@draftmode=0
540     \let\mmz@action\mmz@memoize
541     \fi
542 \fi
543 \else
544     \mmz@trace@process@cmemo@ok

```

If the c-memo was fine, the formal action decided upon is to try utilizing the cc-memo. If it exists and everything is fine with it, `\mmz@process@ccmemo` (defined in section 3.5) will utilize it, i.e. the core of the cc-memo (the part following `\mmzMemo`) will be executed (typically including the single `extern`). Otherwise, `\mmz@process@ccmemo` will trigger either memoization (in the normal mode) or regular compilation (in the readonly mode). This final decision is left to `\mmz@process@ccmemo` because if we made it here, the code would get complicated, as the cc-memo must be processed outside the `\Memoize` group and all the conditionals in this macro.

```

545     \let\mmz@action\mmz@process@ccmemo
546     \fi
547 \fi
548 \fi
549 \fi
550 \fi
551 \mmz@action
552 }

```

`\mmz@compile` This macro performs regular compilation — this is signalled to the memoized code and the memoization driver by setting `\ifinmemoize` to true for the duration of the compilation; `\ifmemoizing` is not touched. The group opened prior to the invocation of `\Memoize` is closed before executing the code in `\mmz@exec@source`, so that compiling the code has the same local effect as if was not submitted to memoization; it is closing this group early which complicates the restoration of `\ifinmemoize` at the end of compilation. Note that `\mmz@exec@source` is already set to properly deal with the current verbatim mode, so any further inspection of `\ifmmz@verbatim` is unnecessary; the same goes for `\ifmmz@ignorespaces`, which was (or at least should be) taken care of by whoever called `\Memoize`.

```

553 \def\mmz@compile{%
554     \mmz@trace@compile
555     \expanded{%
556         \endgroup

```

```

557 \noexpand\inmemoizetrue
558 \the\mmz@exec@source
559 \ifinmemoize\noexpand\inmemoizetrue\else\noexpand\inmemoizefalse\fi
560 }%
561 }

```

`\abortOnError` In Lua_T_EX, we can whether an error occurred during memoization, and abort if it `\mmz@lua@atbeginmemoization` did. (We're going through `memoize.abort`, because `tex.print` does not seem to `\mmz@lua@atendmemoization` work during error handling.) We omit all this in ConT_EXt, as it appears to stop on any error?

```

562 <(!context>
563 \ifdefined\luatexversion
564 \directlua{%
565   luatexbase.add_to_callback(
566     "show_error_message",
567     function()
568       memoize.abort = true
569       texio.write_nl(status.lasterrorstring)
570     end,
571     "Abort memoization on error"
572   )
573 }%
574 \def\mmz@lua@atbeginmemoization{%
575   \directlua{memoize.abort = false}%
576 }%
577 \def\mmz@lua@atendmemoization{%
578   \directlua{%
579     if memoize.abort then
580       tex.print("\noexpand\\mmzAbort")
581     end
582   }%
583 }%
584 \else
585 </!context>
586 \let\mmz@lua@atbeginmemoization\relax
587 \let\mmz@lua@atendmemoization\relax
588 <(!context>\fi

```

`\mmz@memoize` This macro performs memoization — this is signalled to the memoized code and the memoization driver by setting both `\ifinmemoize` and `\ifinmemoizing` to true.

```

589 \def\mmz@memoize{%
590   \mmz@trace@memoize
591   \memoizingtrue
592   \inmemoizetrue

```

Initialize the various macros and registers used in memoization (to be described below, or later). Note that most of these are global, as they might be adjusted arbitrarily deep within the memoized code.

```

593 \edef\memoizinggrouplevel{\the\currentgrouplevel}%
594 \global\mmz@abortfalse
595 \global\mmzUnmemoizablefalse
596 \global\mmz@seq 0
597 \global\setbox\mmz@tbe@box\vbox{}%
598 \global\mmz@ccmemo@resources{}%
599 \global\mmzCMemo{}%
600 \global\mmzCCMemo{}%
601 \global\mmzContextExtra{}%
602 \gdef\mmzAtEndMemoizationExtra{}%
603 \gdef\mmzAfterMemoizationExtra{}%
604 \mmz@lua@atbeginmemoization

```

Execute the pre-memoization hook, the memoized code (wrapped in the driver), and the post-memoization hook.

```
605 \mmzAtBeginMemoization
606 \mmzDriver{\the\mmz@exec@source}%
607 \mmzAtEndMemoization
608 \mmzAtEndMemoizationExtra
609 \mmz@lua@atendmemoization
610 \ifmmzUnmemoizable
```

To permanently prevent memoization, we have to write down the c-memo (containing `\mmzUnmemoizabletrue`). We don't need the extra context in this case.

```
611 \global\mmzContextExtra{}%
612 \gtoksapp\mmzCMemo{\global\mmzUnmemoizabletrue}%
613 \mmz@write@cmemo
614 \mmz@trace@endmemoize@unmemoizable
615 \PackageInfo{memoize}{Marking this code as unmemoizable}%
616 \else
617 \ifmmz@abort
```

If memoization was aborted, we create an empty c-memo, to make sure that no leftover c-memo tricks Memoize into thinking that the code was successfully memoized.

```
618 \mmz@trace@endmemoize@aborted
619 \PackageInfo{memoize}{Memoization was aborted}%
620 \mmz@compute@context@mdfivesum
621 \mmz@write@cmemo
622 \else
```

If memoization was not aborted, we compute the $\langle context\ md5sum \rangle$, open and write out the memos, and shipout the externs (as pages into the document).

```
623 \mmz@compute@context@mdfivesum
624 \mmz@write@cmemo
625 \mmz@write@ccmemo
626 \mmz@shipout@externs
627 \mmz@trace@endmemoize@ok
628 \fi
629 \fi
```

After closing the group, we execute the final, after-memoization hook (we pre-expand the regular macro; the extra macro was assigned to globally). In the after-memoization code, `\mmzIncludeExtern` points to a macro which can include the extern from `\mmz@tbe@box`, which makes it possible to typeset the extern by dropping the contents of `\mmzCCMemo` into this hook — but note that this will only work if `\ifmmzkeepexterns` was in effect at the end of memoization.

```
630 \expandafter\endgroup
631 \expandafter\let
632 \expandafter\mmzIncludeExtern\expandafter\mmz@include@extern@from@tbe@box
633 \mmzAfterMemoization
634 \mmzAfterMemoizationExtra
635 }
```

`\memoizinggrouplevel` This macro stores the group level at the beginning of memoization. It is deployed by `\IfMemoizing`, normally used by integrated drivers.

```
636 \def\memoizinggrouplevel{-1}%
```

`\mmzAbort` Memoized code may execute this macro to abort memoization.

```
637 \def\mmzAbort{\global\mmz@aborttrue}
```

`\ifmmz@abort` This conditional serves as a signal that something went wrong during memoization (where it is set to true by `\mmzAbort`), or c(c)-memo processing. The assignment to this conditional should always be global (because it may be set during memoization).

```
638 \newif\ifmmz@abort
```

`\mmzUnmemoizable` Memoized code may execute `\mmzUnmemoizable` to abort memoization and mark (in the c-memo) that memoization should never be attempted again. The c-memo is composed by `\mmz@memoize`.

```
639 \def\mmzUnmemoizable{\global\mmzUnmemoizabletrue}
```

`\ifmmzUnmemoizable` This conditional serves as a signal that the code should never be memoized. It can be set (a) during memoization (that's why it should be assigned globally), after which it is inspected by `\mmz@memoize`, and (b) from the c-memo, in which case it is inspected by `\Memoize`.

```
640 \newif\ifmmzUnmemoizable
```

`\mmzAtBeginMemoization` The memoization hooks and their keys. The hook macros may be set either before or during memoization. In the former case, one should modify the primary `\mmzAtEndMemoization` macro (`\mmzAtBeginMemoization`, `\mmzAtEndMemoization`, `\mmzAfterMemoization`), at begin memoization and the assignment should be local. In the latter case, one should modify the extra `\mmzAtEndMemoizationExtra`, `\mmzAfterMemoizationExtra`; there is no `\mmzAtBeginMemoizationExtra`, and the assignment should be global. The keys automatically adapt to the situation, by appending either to the primary or the the extra macro; if at begin memoization is used during memoization, the given code is executed immediately. We will use this “extra” approach and the auto-adapting keys for other options, like `context`, as well.

```
641 \def\mmzAtBeginMemoization{}
642 \def\mmzAtEndMemoization{}
643 \def\mmzAfterMemoization{}
644 \mmzset{
645   at begin memoization/.code={%
646     \ifmemoizing
647       \expandafter\@firstofone
648     \else
649       \expandafter\appto\expandafter\mmzAtBeginMemoization
650     \fi
651     {#1}%
652   },
653   at end memoization/.code={%
654     \ifmemoizing
655       \expandafter\gappto\expandafter\mmzAtEndMemoizationExtra
656     \else
657       \expandafter\appto\expandafter\mmzAtEndMemoization
658     \fi
659     {#1}%
660   },
661   after memoization/.code={%
662     \ifmemoizing
663       \expandafter\gappto\expandafter\mmzAfterMemoizationExtra
664     \else
665       \expandafter\appto\expandafter\mmzAfterMemoization
666     \fi
667     {#1}%
668   },
669 }
```

`driver` This key sets the (formal) memoization driver. The function of the driver is to produce the memos and externs while executing the submitted code.

```

670 \mmzset{
671   driver/.store in=\mmzDriver,
672   driver=\mmzSingleExternDriver,
673 }

```

\ifmmzkeepexterns This conditional causes Memoize not to empty out `\mmz@tbe@box`, holding the externs collected during memoization, while shipping them out.

```

674 \newif\ifmmzkeepexterns

```

\mmzSingleExternDriver The default memoization driver externalizes the submitted code. It always produces exactly one extern, and including the extern will be the only effect of inputting the cc-memo (unless the memoized code contained some commands, like `\label`, which added extra instructions to the cc-memo.) The macro (i) adds `\quitvmode` to the cc-memo, if we're capturing into a horizontal box, and it puts it to the very front, so that it comes before any `\label` and `\index` replications, guaranteeing (hopefully) that they refer to the correct page; (ii) takes the code and typesets it in a box (`\mmz@box`); (iii) submits the box for externalization; (iv) adds the extern-inclusion code to the cc-memo, and (v) puts the box into the document (again prefixing it with `\quitvmode` if necessary). (The listing region markers help us present this code in the manual.)

```

675 \long\def\mmzSingleExternDriver#1{%
676   \xtoksapp\mmzCCMemo{\mmz@maybe@quitvmode}%
677   \setbox\mmz@box\mmz@capture{#1}%
678   \mmzExternalizeBox\mmz@box\mmz@temptoks
679   \xtoksapp\mmzCCMemo{\the\mmz@temptoks}%
680   \mmz@maybe@quitvmode\box\mmz@box
681 }

```

capture The default memoization driver uses `\mmz@capture` and `\mmz@maybe@quitvmode`, which are set by this key. `\mmz@maybe@quitvmode` will be expanded, but for \TeX , we have defined `\quitvmode` as a synonym for `\leavevmode`, which is a macro rather than a primitive, so we have to prevent its expansion in that case. It is easiest to just add `\noexpand`, regardless of the engine used.

```

682 \mmzset{
683   capture/.is choice,
684   capture/hbox/.code={%
685     \let\mmz@capture\hbox
686     \def\mmz@maybe@quitvmode{\noexpand\quitvmode}%
687   },
688   capture/vbox/.code={%
689     \let\mmz@capture\vbox
690     \def\mmz@maybe@quitvmode{}%
691   },
692   capture=hbox,
693 }

```

The memoized code may be memoization-aware; in such a case, we say that the driver is *integrated* into the code. Code containing an integrated driver must take care to execute it only when memoizing, and not during a regular compilation. The following key and macro can help here, see [M§4.4.4](#) for details.

integrated driver This is an advice key, residing in `/mmz/auto`. Given $\langle suffix \rangle$ as the only argument, it declares conditional `\ifmemoizing<suffix>`, and sets the driver for the automemoized command to a macro which sets this conditional to true. The declared conditional is *internal* and should not be used directly, but only via `\IfMemoizing` — because it will not be declared when package NoMemoize or only Memoizable is loaded.

```

694 \mmzset{
695   auto/integrated driver/.style={
696     after setup={\expandafter\newif\csname ifmmz@memoizing#1\endcsname},
697     driver/.expand once={%
698       \csname mmz@memoizing#1true\endcsname

```

Without this, we would introduce an extra group around the memoized code.

```
699     \@firstofone
700   }%
701 },
702 }
```

\IfMemoizing Without the optional argument, the condition is satisfied when the internal conditional `\ifmemoizing<suffix>`, declared by `integrated driver`, is true. With the optional argument `<offset>`, the current group level must additionally match the memoizing group level, modulo `<offset>` — this makes sure that the conditional comes out as false in a regular compilation embedded in a memoization.

```
703 \newcommand\IfMemoizing[2][\mmz@ifmemoizing@nogrouplevel]{%>\fi
704 \csname ifmmz@memoizing#2\endcsname%>\if
```

One `\relax` is for the `\numexpr`, another for `\ifnum`. Complications arise when `#1` is the optional argument default (defined below). In that case, the content of `\mmz@ifmemoizing@nogrouplevel` closes off the `\ifnum` conditional (with both the true and the false branch empty), and opens up a new one, `\iftrue`. Effectively, we're not testing for the group level match.

```
705 \ifnum\currentgrouplevel=\the\numexpr\memoizinggrouplevel+#1\relax\relax
706   \expandafter\expandafter\expandafter\@firstoftwo
707 \else
708   \expandafter\expandafter\expandafter\@secondoftwo
709 \fi
710 \else
711   \expandafter\@secondoftwo
712 \fi
713 }
714 \def\mmz@ifmemoizing@nogrouplevel{0\relax\relax\fi\iftrue}
```

Tracing We populate the hooks which send the tracing info to the terminal.

```
715 \def\mmz@trace#1{\advice@typeout{[tracing memoize] #1}}
716 \def\mmz@trace@context{\mmz@trace{\space\space
717   Context: "\expandonce{\mmz@context@key}" --> \mmz@context@mdfivesum}}
718 \def\mmz@trace@Memoize@on{%
719   \mmz@trace{%
720     Entering \noexpand\Memoize (%
721     \ifmemoize enabled\else disabled\fi,
722     \ifnum\mmz@mode=\mmz@mode@recompile recompile\fi
723     \ifnum\mmz@mode=\mmz@mode@readonly readonly\fi
724     \ifnum\mmz@mode=\mmz@mode@normal normal\fi
725     \space mode) on line \the\inputlineno
726   }%
727   \mmz@trace{\space\space Code: \the\mmz@mdfive@source}%
728 }
729 \def\mmz@trace@code@mdfive@on{\mmz@trace{\space\space
730   Code md5sum: \mmz@code@mdfivesum}}
731 \def\mmz@trace@compile@on{\mmz@trace{\space\space Compiling}}
732 \def\mmz@trace@memoize@on{\mmz@trace{\space\space Memoizing}}
733 \def\mmz@trace@endmemoize@ok@on{\mmz@trace{\space\space
734   Memoization completed}}%
735 \def\mmz@trace@endmemoize@aborted@on{\mmz@trace{\space\space
736   Memoization was aborted}}
737 \def\mmz@trace@endmemoize@unmemoizable@on{\mmz@trace{\space\space
738   Marking this code as unmemoizable}}
```

No need for `\mmz@trace@endmemoize@fail`, as abortion results in a package warning anyway.

```
739 \def\mmz@trace@process@cmemo@on{\mmz@trace{\space\space
740   Attempting to utilize c-memo \mmz@cmemo@path}}
```

```

741 \def\mmz@trace@process@no@cmemo@on{\mmz@trace{\space\space
742   C-memo does not exist}}
743 \def\mmz@trace@process@cmemo@ok@on{\mmz@trace{\space\space
744   C-memo was processed successfully}\mmz@trace@context}
745 \def\mmz@trace@process@cmemo@fail@on{\mmz@trace{\space\space
746   C-memo input failed}}
747 \def\mmz@trace@cmemo@unmemoizable@on{\mmz@trace{\space\space
748   This code was marked as unmemoizable}}
749 \def\mmz@trace@process@ccmemo@on{\mmz@trace{\space\space
750   Attempting to utilize cc-memo \mmz@ccmemo@path\space
751   (\ifmmz@direct@ccmemo@input\else in\fi direct input)}}
752 \def\mmz@trace@resource@on#1{\mmz@trace{\space\space
753   Extern file does not exist: #1}}
754 \def\mmz@trace@process@ccmemo@ok@on{%
755   \mmz@trace{\space\space Utilization successful}}
756 \def\mmz@trace@process@no@ccmemo@on{%
757   \mmz@trace{\space\space CC-memo does not exist}}
758 \def\mmz@trace@process@ccmemo@fail@on{%
759   \mmz@trace{\space\space Cc-memo input failed}}

```

tracing The user interface for switching the tracing on and off; initially, it is off. Note that there is no underlying conditional. The off version simply \lets all the tracing hooks to \relax, so that the overhead of having the tracing functionality available is negligible.

```

760 \mmzset{%
761   trace/.is choice,
762   trace/.default=true,
763   trace/true/.code=\mmzTracingOn,
764   trace/false/.code=\mmzTracingOff,
765 }
766 \def\mmzTracingOn{%
767   \let\mmz@trace@Memoize\mmz@trace@Memoize@on
768   \let\mmz@trace@code@mdfive\mmz@trace@code@mdfive@on
769   \let\mmz@trace@compile\mmz@trace@compile@on
770   \let\mmz@trace@memoize\mmz@trace@memoize@on
771   \let\mmz@trace@process@cmemo\mmz@trace@process@cmemo@on
772   \let\mmz@trace@endmemoize@ok\mmz@trace@endmemoize@ok@on
773   \let\mmz@trace@endmemoize@unmemoizable\mmz@trace@endmemoize@unmemoizable@on
774   \let\mmz@trace@endmemoize@aborted\mmz@trace@endmemoize@aborted@on
775   \let\mmz@trace@process@cmemo\mmz@trace@process@cmemo@on
776   \let\mmz@trace@process@cmemo@ok\mmz@trace@process@cmemo@ok@on
777   \let\mmz@trace@process@no@cmemo\mmz@trace@process@no@cmemo@on
778   \let\mmz@trace@process@cmemo@fail\mmz@trace@process@cmemo@fail@on
779   \let\mmz@trace@cmemo@unmemoizable\mmz@trace@cmemo@unmemoizable@on
780   \let\mmz@trace@process@ccmemo\mmz@trace@process@ccmemo@on
781   \let\mmz@trace@resource\mmz@trace@resource@on
782   \let\mmz@trace@process@ccmemo@ok\mmz@trace@process@ccmemo@ok@on
783   \let\mmz@trace@process@no@ccmemo\mmz@trace@process@no@ccmemo@on
784   \let\mmz@trace@process@ccmemo@fail\mmz@trace@process@ccmemo@fail@on
785 }
786 \def\mmzTracingOff{%
787   \let\mmz@trace@Memoize\relax
788   \let\mmz@trace@code@mdfive\relax
789   \let\mmz@trace@compile\relax
790   \let\mmz@trace@memoize\relax
791   \let\mmz@trace@process@cmemo\relax
792   \let\mmz@trace@endmemoize@ok\relax
793   \let\mmz@trace@endmemoize@unmemoizable\relax
794   \let\mmz@trace@endmemoize@aborted\relax
795   \let\mmz@trace@process@cmemo\relax
796   \let\mmz@trace@process@cmemo@ok\relax
797   \let\mmz@trace@process@no@cmemo\relax
798   \let\mmz@trace@process@cmemo@fail\relax

```



```

799 \let\mmz@trace@cmemo@unmemoizable\relax
800 \let\mmz@trace@process@ccmemo\relax
801 \let\mmz@trace@resource\@gobble
802 \let\mmz@trace@process@ccmemo@ok\relax
803 \let\mmz@trace@process@no@ccmemo\relax
804 \let\mmz@trace@process@ccmemo@fail\relax
805 }
806 \mmzTracingOff

```

3.3 Context

`\mmzContext` The context expression is stored in two token registers. Outside memoization, we will locally
`\mmzContextExtra` assign to `\mmzContext`; during memoization, we will globally assign to `\mmzContextExtra`.

```

807 \newtoks\mmzContext
808 \newtoks\mmzContextExtra

```

`context` The user interface keys for context manipulation hide the complexity underlying the context
`clear context` storage from the user.

```

809 \mmzset{%
810   context/.code={%
811     \ifmemoizing
812       \expandafter\gtoksapp\expandafter\mmzContextExtra
813     \else
814       \expandafter\toksapp\expandafter\mmzContext
815     \fi

```

We append a comma to the given context chunk, for disambiguation.

```

816   {#1,}%
817 },
818 clear context/.code={%
819   \ifmemoizing
820     \expandafter\global\expandafter\mmzContextExtra
821   \else
822     \expandafter\mmzContext
823   \fi
824   {}}%
825 },
826 clear context/.value forbidden,

```

`meaning to context` Utilities to put the meaning of various stuff into context.

`csname meaning to context`

`key meaning to context`

`key value to context`

`/handlers/.meaning to context`

`/handlers/.value to context`

```

827 meaning to context/.code={\forcsvlist\mmz@mtoc{#1}},
828 csname meaning to context/.code={\mmz@mtoc@csname{#1}},
829 key meaning to context/.code={%
830   \forcsvlist\mmz@mtoc\mmz@mtoc@keycmd{#1}},
831 key value to context/.code={\forcsvlist\mmz@mtoc@key{#1}},
832 /handlers/.meaning to context/.code={\expanded{%
833   \noexpand\mmz@mtoc@csname{pgfk@\pgfkeyscurrentpath/.@cmd}}},
834 /handlers/.value to context/.code={%
835   \expanded{\noexpand\mmz@mtoc@csname{pgfk@\pgfkeyscurrentpath}}},
836 }

```

```

837 \def\mmz@mtoc#1{%
838   \collargs@cs@cases{#1}%
839   {\mmz@mtoc@cmd{#1}}%
840   {\mmz@mtoc@error@notcsorenv{#1}}%
841   {%
842     \mmz@mtoc@csname{%
843 <context>      start%
844                #1}%

```



```

845      \mmz@mtoc@csname{%
846 <latex, plain>      end%
847 <context>          stop%
848                  #1}%
849      }%
850  }
851  \def\mmz@mtoc@cmd#1{%
852      \begingroup
853      \escapechar=-1
854      \expandafter\endgroup
855      \expandafter\mmz@mtoc@csname\expandafter{\string#1}%
856  }
857  \def\mmz@mtoc@csname#1{%
858      \pgfkeysvalueof{/mmz/context/.@cmd}%
859      \detokenize{#1}={\expandafter\meaning\csname#1\endcsname}%
860      \pgfeov
861  }
862  \def\mmz@mtoc@key#1{\mmz@mtoc@csname{pgfk@#1}}
863  \def\mmz@mtoc@keycmd#1{\mmz@mtoc@csname{pgfk@#1/.@cmd}}
864  \def\mmz@mtoc@error@notcsorenv#1{%
865      \PackageError{memoize}{'\detokenize{#1}' passed to key 'meaning to context'
866          is neither a command nor an environment}{}%
867  }

```

3.4 C-memos

The path to a c-memo consists of the path prefix, the MD5 sum of the memoized code, and suffix `.memo`.

```
868 \def\mmz@cmemo@path{\mmz@prefix\mmz@code@mdfivesum.memo}
```

`\mmzCMemo` The additional, free-form content of the c-memo is collected in this token register.

```
869 \newtoks\mmzCMemo
```

`include source in cmemo` Should we include the memoized code in the c-memo? By default, yes.

```
\ifmmz@include@source
```

```

870 \mmzset{%
871     include source in cmemo/.is if=mmz@include@source,
872 }
873 \newif\ifmmz@include@source
874 \mmz@include@sourcetrue

```

`\mmz@write@cmemo` This macro creates the c-memo from the contents of `\mmzContextExtra` and `\mmzCMemo`.

```
875 \def\mmz@write@cmemo{%
```

Open the file for writing.

```
876 \immediate\openout\mmz@out{\mmz@cmemo@path}%
```

The memo starts with the `\mmzMemo` marker (a signal that the memo is valid).

```
877 \immediate\write\mmz@out{\noexpand\mmzMemo}%
```

We store the content of `\mmzContextExtra` by writing out a command that will (globally) assign its content back into this register.

```

878 \immediate\write\mmz@out{%
879     \global\mmzContextExtra{\the\mmzContextExtra}\collargs@percentchar
880 }%

```

Write out the free-form part of the c-memo.

```
881 \immediate\write\mmz@out{\the\mmzCMemo\collargs@percentchar}%
```

When `include source` in `cmemo` is in effect, add the memoized code, hiding it behind the `\mmzSource` marker.

```
882 \ifmmz@include@source
883   \immediate\write\mmz@out{\noexpand\mmzSource}%
884   \immediate\write\mmz@out{\the\mmz@mdfive@source}%
885 \fi
```

Close the file.

```
886 \immediate\closeout\mmz@out
```

Record that we wrote a new c-memo.

```
887 \pgfkeysalso{/mmz/record/new cmemo={\mmz@cmemo@path}}%
888 }
```

`\mmzSource` The c-memo memoized code marker. This macro is synonymous with `\endinput`, so the source following it is ignored when inputting the c-memo.

```
889 \let\mmzSource\endinput
```

`\mmz@process@cmemo` This macro inputs the c-memo, which will update the context code, which we can then compute the MD5 sum of.

```
890 \def\mmz@process@cmemo{%
891   \mmz@trace@process@cmemo
```

`\ifmmz@abort` serves as a signal that the c-memo exists and is of correct form.

```
892   \global\mmz@aborttrue
```

If c-memo sets `\ifmmzUnmemoizable`, we will compile regularly.

```
893   \global\mmzUnmemoizablefalse
894   \def\mmzMemo{\global\mmz@abortfalse}%
```

Just a safeguard ... c-memo assigns to `\mmzContextExtra` anyway.

```
895   \global\mmzContextExtra{}}%
```

Input the c-memo, if it exists, and record that we have used it.

```
896   \IfFileExists{\mmz@cmemo@path}{%
897     \input{\mmz@cmemo@path}%
898     \pgfkeysalso{/mmz/record/used cmemo={\mmz@cmemo@path}}%
899   }{%
900     \mmz@trace@process@no@cmemo
901   }%
```

Compute the context MD5 sum.

```
902   \mmz@compute@context@mdfivesum
903 }
```

`\mmz@compute@context@mdfivesum` This macro computes the MD5 sum of the concatenation of `\mmzContext` and `\mmzContextExtra`, and writes out the tracing info when `trace context` is in effect. The argument is the tracing note.

```
904 \def\mmz@compute@context@mdfivesum{%
905   \xdef\mmz@context@key{\the\mmzContext\the\mmzContextExtra}%
```

A special provision for padding, which occurs in the context by default, and may contain otherwise undefined macros referring to the extern dimensions. We make sure that when we expand the context key, `\mmz@paddings` contains the stringified `\width` etc., while these macros (which may be employed by the end user in the context expression), are returned to their original definitions.

```

906 \begingroup
907 \begingroup
908 \def\width{\string\width}%
909 \def\height{\string\height}%
910 \def\depth{\string\depth}%
911 \edef\mmz@paddings{\mmz@paddings}%
912 \expandafter\endgroup
913 \expandafter\def\expandafter\mmz@paddings\expandafter{\mmz@paddings}%

```

We pre-expand the concatenated context, for tracing/inclusion in the cc-memo. In \LaTeX , we protect the expansion, as the context expression may contain whatever.

```

914 <latex> \protected@xdef
915 <!latex> \xdef
916 \mmz@context@key{\mmz@context@key}%
917 \endgroup

```

Compute the MD5 sum. We have to assign globally, because this macro is (also) called after inputting the c-memo, while the resulting MD5 sum is used to input the cc-memo, which happens outside the `\Memoize` group. `\mmz@context@mdfivesum`.

```

918 \xdef\mmz@context@mdfivesum{\pdfmdfivesum{\expandonce\mmz@context@key}}%
919 }

```

3.5 Cc-memos

The path to a cc-memo consists of the path prefix, the hyphen-separated MD5 sums of the memoized code and the (evaluated) context, and suffix `.memo`.

```

920 \def\mmz@ccmemo@path{%
921 \mmz@prefix\mmz@code@mdfivesum-\mmz@context@mdfivesum.memo}

```

The structure of a cc-memo:

- the list of resources consisting of calls to `\mmzResource`;
- the core memo code (which includes the externs when executed), introduced by marker `\mmzMemo`; and,
- optionally, the context expansion, introduced by marker `\mmzThisContext`.

We begin the cc-memo with a list of extern files included by the core memo code so that we can check whether these files exist prior to executing the core memo code. Checking this on the fly, while executing the core memo code, would be too late, as that code is arbitrary (and also executed outside the `\Memoize` group).

`\mmzCCMemo` During memoization, the core content of the cc-memo is collected into this token register.

```

922 \newtoks\mmzCCMemo

```

`include context in ccmemo` Should we include the context expansion in the cc-memo? By default, no.

```
\ifmmz@include@context
```

```

923 \newif\ifmmz@include@context
924 \mmzset{%
925 include context in ccmemo/.is if=mmz@include@context,
926 }

```

`direct ccmemo input` When this conditional is false, the cc-memo is read indirectly, via a token register, `\ifmmz@direct@ccmemo@input` to facilitate inverse search.

```
927 \newif\ifmmz@direct@ccmemo@input
928 \mmzset{%
929   direct ccmemo input/.is if=mmz@direct@ccmemo@input,
930 }
```

`\mmz@write@ccmemo` This macro creates the cc-memo from the list of resources in `\mmz@ccmemo@resources` and the contents of `\mmzCCMemo`.

```
931 \def\mmz@write@ccmemo{%
```

Open the cc-memo file for writing. Note that the filename contains the context MD5 sum, which can only be computed after memoization, as the memoized code can update the context. This is one of the two reasons why we couldn't write the cc-memo directly into the file, but had to collect its contents into token register `\mmzCCMemo`.

```
932   \immediate\openout\mmz@out{\mmz@ccmemo@path}%
```

Token register `\mmz@ccmemo@resources` consists of calls to `\mmz@ccmemo@append@resource`, so the following code writes down the list of created externs into the cc-memo. Wanting to have this list at the top of the cc-memo is the other reason for the roundabout creation of the cc-memo — the resources become known only during memoization, as well.

```
933   \begingroup
934   \the\mmz@ccmemo@resources
935   \endgroup
```

Write down the content of `\mmzMemo`, but first introduce it by the `\mmzMemo` marker.

```
936   \immediate\write\mmz@out{\noexpand\mmzMemo}%
937   \immediate\write\mmz@out{\the\mmzCCMemo\collargs@percentchar}%
```

Write down the context tracing info when `include context in ccmemo` is in effect.

```
938   \ifmmz@include@context
939     \immediate\write\mmz@out{\noexpand\mmzThisContext}%
940     \immediate\write\mmz@out{\expandonce{\mmz@context@key}}%
941   \fi
```

Insert the end-of-file marker and close the file.

```
942   \immediate\write\mmz@out{\noexpand\mmzEndMemo}%
943   \immediate\closeout\mmz@out
```

Record that we wrote a new cc-memo.

```
944   \pgfkeysalso{/mmz/record/new ccmemo={\mmz@ccmemo@path}}%
945 }
```

`\mmz@ccmemo@append@resource` Append the resource to the cc-memo (we are nice to external utilities and put each resource on its own line). #1 is the sequential number of the extern belonging to the memoized code; below, we assign it to `\mmz@seq`, which appears in `\mmz@extern@name`. Note that `\mmz@extern@name` only contains the extern filename — without the path, so that externs can be used by several projects, or copied around.

```
946 \def\mmz@ccmemo@append@resource#1{%
947   \mmz@seq=#1\relax
948   \immediate\write\mmz@out{%
949     \string\mmzResource{\mmz@extern@name}\collargs@percentchar}%
950 }
```

`\mmzResource` A list of these macros is located at the top of a cc-memo. The macro checks for the existence of the extern file, given as #1. If the extern does not exist, we redefine `\mmzMemo` to `\endinput`, so that the core content of the cc-memo is never executed; see also `\mmz@process@ccmemo` above.

```
951 \def\mmzResource#1{%
```

We check for existence using `\pdffilesize`, because an empty PDF, which might be produced by a failed \TeX -based extraction, should count as no file. The 0 behind `\ifnum` is there because `\pdffilesize` returns an empty string when the file does not exist.

```
952 \ifnum0\pdf@filesize{\mmz@prefix@dir#1}=0
953 \ifmmz@direct@ccmemo@input
954 \let\mmzMemo\endinput
955 \else
```

With indirect cc-memo input, we simulate end-of-input by grabbing everything up to the end-of-memo marker. In the indirect cc-memo input, a `\par` token shows up after `\mmzEndMemo`, I'm not sure why (`\everyeof={}` does not help).

```
956 \long\def\mmzMemo##1\mmzEndMemo\par{}%
957 \fi
958 \mmz@trace@resource{#1}%
959 \fi
960 }
```

`\mmz@process@ccmemo` This macro processes the cc-memo.

`\mmzThisContext`

```
\mmzEndMemo 961 \def\mmz@process@ccmemo{%
962 \mmz@trace@process@ccmemo
```

The following conditional signals whether cc-memo was successfully utilized. If the cc-memo file does not exist, `\ifmmz@abort` will remain true. If it exists, it is headed by the list of resources. If a resource check fails, `\mmzMemo` (which follows the list of resources) is redefined to `\endinput`, so `\ifmmz@abort` remains true. However, if all resource checks are successful, `\mmzMemo` marker is reached with the below definition in effect, so `\ifmmz@abort` becomes false. Note that this marker also closes the `\Memoize` group, so that the core cc-memo content is executed in the original group — and that this does not happen if anything goes wrong!

```
963 \global\mmz@aborttrue
```

Note that `\mmzMemo` may be redefined by `\mmzResource` upon an unavailable extern file.

```
964 \def\mmzMemo{%
965 \endgroup
966 \global\mmz@abortfalse
```

We `\let` the control sequence used for extern inclusion in the cc-memo to the macro which includes the extern from the extern file.

```
967 \let\mmzIncludeExtern\mmz@include@extern
968 }%
```

Define `\mmzEndMemo` wrt `\ifmmz@direct@ccmemo@input`, whose value will be lost soon because `\mmMemo` will close the group — that's also why this definition is global.

```
969 \xdef\mmzEndMemo{%
970 \ifmmz@direct@ccmemo@input
971 \noexpand\endinput
972 \else
```

In the indirect cc-memo input, a `\par` token shows up after `\mmzEndMemo`, I'm not sure why (`\everyeof={}` does not help).

```
973 \unexpanded{%
```

```

974     \def\mmz@temp\par{}%
975     \mmz@temp
976   }%
977   \fi
978 }%

```

The cc-memo context marker, again wrt `\ifmmz@direct@ccmemo@input` and globally. With direct cc-memo input, this macro is synonymous with `\endinput`, so the (expanded) context following it is ignored when inputting the cc-memo. With indirect input, we simulate end-of-input by grabbing everything up to the end-of-memo marker (plus gobble the `\par` mentioned above).

```

979   \xdef\mmzThisContext{%
980     \ifmmz@direct@ccmemo@input
981       \noexpand\endinput
982     \else
983       \unexpanded{%
984         \long\def\mmz@temp##1\mmzEndMemo\par{}%
985         \mmz@temp
986       }%
987     \fi
988 }%

```

Input the cc-memo if it exists.

```

989   \IfFileExists{\mmz@ccmemo@path}{%
990     \ifmmz@direct@ccmemo@input
991       \input{\mmz@ccmemo@path}%
992     \else

```

Indirect cc-memo input reads the cc-memo into a token register and executes the contents of this register.

```

993     \filetotoks\toks@{\mmz@ccmemo@path}%
994     \the\toks@
995   \fi

```

Record that we have used the cc-memo.

```

996     \pgfkeysalso{/mmz/record/used ccmemo={\mmz@ccmemo@path}}%
997   }{%
998     \mmz@trace@process@no@ccmemo
999   }%
1000   \ifmmz@abort

```

The cc-memo doesn't exist, or some of the resources don't. We need to memoize, but we'll do it only if `readonly` is not in effect, otherwise we'll perform a regular compilation. (Note that we are still in the group opened prior to executing `\Memoize`.)

```

1001     \mmz@trace@process@ccmemo@fail
1002     \ifnum\mmz@mode=\mmz@mode@readonly\relax
1003       \expandafter\expandafter\expandafter\mmz@compile
1004     \else
1005       \expandafter\expandafter\expandafter\mmz@memoize
1006     \fi
1007   \else
1008     \mmz@trace@process@ccmemo@ok
1009   \fi
1010 }

```

3.6 The externs

The [path](#) to an extern is like the path to a cc-memo, modulo suffix `.pdf`, of course. However, in case memoization of a chunk produces more than one extern, the filename of any non-first extern

includes `\mmz@seq`, the sequential number of the extern as well (we start the numbering at 0). We will have need for several parts of the full path to an extern: the basename, the filename, the path without the suffix, and the full path.

```

1011 \newcount\mmz@seq
1012 \def\mmz@extern@basename{%
1013   \mmz@prefix@name\mmz@code@mdfivesum-\mmz@context@mdfivesum
1014   \ifnum\mmz@seq>0 -\the\mmz@seq\fi
1015 }
1016 \def\mmz@extern@name{\mmz@extern@basename.pdf}
1017 \def\mmz@extern@basepath{\mmz@prefix@dir\mmz@extern@basename}
1018 \def\mmz@extern@path{\mmz@extern@basepath.pdf}

```

padding left These options set the amount of space surrounding the bounding box of the externalized graphics
padding right in the resulting PDF, i.e. in the extern file. This allows the user to deal with TikZ overlays,
padding top `\rlap` and `\llap`, etc.

padding bottom

```

1019 \mmzset{
1020   padding left/.store in=\mmz@padding@left,
1021   padding right/.store in=\mmz@padding@right,
1022   padding top/.store in=\mmz@padding@top,
1023   padding bottom/.store in=\mmz@padding@bottom,

```

padding A shortcut for setting all four paddings at once.

```

1024   padding/.style={
1025     padding left=#1, padding right=#1,
1026     padding top=#1, padding bottom=#1
1027   },

```

The default padding is what pdfTeX puts around the page anyway, 1 inch, but we'll use `1 in` rather than `1 true in`, which is the true default value of `\pdfhorigin` and `\pdfvorigin`, as we want the padding to adjust with magnification.

```

1028   padding=1in,

```

padding to context This key adds padding to the context. Note that we add the padding expression (`\mmz@paddings`, defined below, refers to all the individual padding macros), not the actual value (at the time of expansion). This is so because `\width`, `\height` and `\depth` are not defined outside extern shipout routines, and the context is evaluated elsewhere.

```

1029   padding to context/.style={
1030     context={padding=(\mmz@paddings)},
1031   },

```

Padding nearly always belongs into the context — the exception being memoized code which produces no externs (^M§4.4.2) — so we execute this key immediately.

```

1032   padding to context,
1033 }
1034 \def\mmz@paddings{%
1035   \mmz@padding@left,\mmz@padding@bottom,\mmz@padding@right,\mmz@padding@top
1036 }

```

\mmzExternalizeBox This macro is the public interface to externalization. In Memoize itself, it is called from the default memoization driver, `\mmzSingleExternDriver`, but it should be called by any driver that wishes to produce an extern, see ^M§4.4 for details. It takes two arguments:

- #1 The box that we want to externalize. It's content will remain intact. The box may be given either as a control sequence, declared via `\newbox`, or as box number (say, 0).

#2 The token register which will receive the code that includes the extern into the document; it is the responsibility of the memoization driver to (globally) include the contents of the register in the cc-memo, i.e. in token register `\mmzCCMemo`. This argument may be either a control sequence, declared via `\newtoks`, or a `\toks`*(token register number)*.

```
1037 \def\mmzExternalizeBox#1#2{%
1038   \begingroup
```

A courtesy to the user, so they can define padding in terms of the size of the externalized graphics.

```
1039   \def\width{\wd#1 }%
1040   \def\height{\ht#1 }%
1041   \def\depth{\dp#1 }%
```

Store the extern-inclusion code in a temporary macro, which will be smuggled out of the group.

```
1042   \xdef\mmz@global@temp{%
```

Executing `\mmzIncludeExtern` from the cc-memo will include the extern into the document.

```
1043     \noexpand\mmzIncludeExtern
```

`\mmzIncludeExtern` identifies the extern by its sequence number, `\mmz@seq`.

```
1044     {\the\mmz@seq}%
```

What kind of box? We `\noexpand` the answer just in case someone redefined them.

```
1045     \ifhbox#1\noexpand\hbox\else\noexpand\vbox\fi
```

The dimensions of the extern.

```
1046     {\the\wd#1}%
1047     {\the\ht#1}%
1048     {\the\dp#1}%
```

The padding values.

```
1049     {\the\dimexpr\mmz@padding@left}%
1050     {\the\dimexpr\mmz@padding@bottom}%
1051     {\the\dimexpr\mmz@padding@right}%
1052     {\the\dimexpr\mmz@padding@top}%
1053   }%
```

Prepend the new extern box into the global extern box where we collect all the externs of this memo. Note that we `\copy` the extern box, retaining its content — we will also want to place the extern box in its regular place in the document.

```
1054   \global\setbox\mmz@tbe@box\vbox{\copy#1\unvbox\mmz@tbe@box}%
```

Add the extern to the list of resources, which will be included at the top of the cc-memo, to check whether the extern files exists at the time the cc-memo is utilized. In the cc-memo, the list will contain full extern filenames, which are currently unknown, but no matter; right now, providing the extern sequence number suffices, the full extern filename will be produced at the end of memoization, once the context MD5 sum is known.

```
1055   \xtoksapp\mmz@ccmemo@resources{%
1056     \noexpand\mmz@ccmemo@append@resource{\the\mmz@seq}%
1057   }%
```

Increment the counter containing the sequence number of the extern within this memo.

```
1058   \global\advance\mmz@seq1
```


Assign the extern-including code into the token register given in #2. This register may be given either as a control sequence or as `\toks<token register number>`, and this is why we have temporarily stored the code (into `\mmz@global@temp`) globally: a local storage with `\expandafter\endgroup\expandafter` here would fail with the receiving token register given as `\toks<token register number>`.

```
1059 \endgroup
1060 #2\expandafter{\mmz@global@temp}%
1061 }
```

`\mmz@ccmemo@resources` This token register, populated by `\mmz@externalize@box` and used by `\mmz@write@ccmemo`, holds the list of externs produced by memoization of the current chunk.

```
1062 \newtoks\mmz@ccmemo@resources
```

`\mmz@tbe@box` `\mmz@externalize@box` does not directly dump the extern into the document (as a special page). Rather, the externs are collected into `\mmz@tbe@box`, whose contents are dumped into the document at the end of memoization of the current chunk. In this way, we guarantee that aborted memoization does not pollute the document.

```
1063 \newbox\mmz@tbe@box
```

`\mmz@shipout@externs` This macro is executed at the end of memoization, when the externs are waiting for us in `\mmz@tbe@box` and need to be dumped into the document. It loops through the contents of `\mmz@tbe@box`,² putting each extern into `\mmz@box` and calling `\mmz@shipout@extern`. Note that the latter macro is executed within the group opened by `\vbox` below.

```
1064 \def\mmz@shipout@externs{%
1065   \global\mmz@seq 0
1066   \setbox\mmz@box\vbox{%
```

Set the macros below to the dimensions of the extern box, so that the user can refer to them in the padding specification (which is in turn used in the page setup in `\mmz@shipout@extern`).

```
1067   \def\width{\wd\mmz@box}%
1068   \def\height{\ht\mmz@box}%
1069   \def\depth{\dp\mmz@box}%
1070   \vskip1pt
1071   \ifmmzkeepexterns\expandafter\unvcopy\else\expandafter\unvbox\fi\mmz@tbe@box
1072   \@whiles\ifdimOpt=\lastskip\fi{%
1073     \setbox\mmz@box\lastbox
1074     \mmz@shipout@extern
1075   }%
1076 }%
1077 }
```

`\mmz@shipout@extern` This macro ships out a single extern, which resides in `\mmz@box`, and records the creation of the new extern.

```
1078 \def\mmz@shipout@extern{%
```

Calculate the expected width and height. We have to do this now, before we potentially adjust the box size and paddings for magnification.

```
1079 \edef\expectedwidth{\the\dimexpr
1080   (\mmz@padding@left) + \wd\mmz@box + (\mmz@padding@right)}%
1081 \edef\expectedheight{\the\dimexpr
1082   (\mmz@padding@top) + \ht\mmz@box + \dp\mmz@box + (\mmz@padding@bottom)}%
```

²The looping code is based on TeX.SE answer tex.stackexchange.com/a/25142/16819 by Bruno Le Floch.

Apply the inverse magnification, if `\mag` is not at the default value. We'll do this in a group, which will last until `shipout`.

```
1083 \begingroup
1084 \ifnum\mag=1000
1085 \else
1086   \mmz@shipout@mag
1087 \fi
```

Setup the geometry of the extern page. In plain $\text{T}_{\text{E}}\text{X}$ and $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$, setting `\pdfpagewidth` and `\pdfpageheight` seems to do the trick of setting the extern page dimensions. In $\text{ConT}_{\text{E}}\text{Xt}$, however, the resulting extern page ends up with the PDF `/CropBox` specification of the current regular page, which is then used (ignoring our `mediabox` requirement) when we're including the extern into the document by `\mmzIncludeExtern`. Typically, this results in a page-sized extern. I'm not sure how to deal with this correctly. In the workaround below, we use Lua function `backends.codeinjections.setupcanvas` to set up page dimensions: we first remember the current page dimensions (`\edef\mmz@temp`), then set up the extern page dimensions (`\expanded{...}`), and finally, after shipping out the extern page, revert to the current page dimensions by executing `\mmz@temp` at the very end of this macro.

```
1088 <plain, latex>
1089 \pdfpagewidth\dimexpr
1090   (\mmz@padding@left) + \wd\mmz@box + (\mmz@padding@right)\relax
1091 \pdfpageheight\dimexpr
1092   (\mmz@padding@top) + \ht\mmz@box + \dp\mmz@box+ (\mmz@padding@bottom)\relax
1093 </plain, latex>
1094 <context>
1095 \edef\mmz@temp{%
1096   \noexpand\directlua{
1097     backends.codeinjections.setupcanvas({
1098       paperwidth=\the\numexpr\pagewidth,
1099       paperheight=\the\numexpr\pageheight
1100     })
1101   }%
1102 }%
1103 \expanded{%
1104   \noexpand\directlua{
1105     backends.codeinjections.setupcanvas({
1106       paperwidth=\the\numexpr\dimexpr
1107         \mmz@padding@left + \wd\mmz@box + \mmz@padding@right\relax,
1108       paperheight=\the\numexpr\dimexpr
1109         \mmz@padding@top + \ht\mmz@box + \dp\mmz@box+ \mmz@padding@bottom\relax
1110     })
1111   }%
1112 }%
1113 </context>
```

We complete the page setup by setting the content offset.

```
1114 \hoffset\dimexpr\mmz@padding@left - \pdfhorigin\relax
1115 \voffset\dimexpr\mmz@padding@top - \pdfvorigin\relax
```

We shipout the extern page using the `\shipout` primitive, so that the extern page is not modified, or even registered, by the shipout code of the format or some package. I can't imagine those shipout routines ever needing to know about the extern page. In fact, most often knowing about it would be undesirable. For example, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ and $\text{ConT}_{\text{E}}\text{Xt}$ count the “real” pages, but usually to know whether they are shipping out an odd or an even page, or to make the total number of pages available to subsequent compilations. Taking the extern pages into account would disrupt these mechanisms.

Another thing: delayed `\writes`. We have to make sure that any $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -style protected stuff in those is not expanded. We don't bother introducing a special group, as we'll close the `\mag` group right after the shipout anyway.

```

1116 <latex> \let\protect\noexpand
1117         \pdf@primitive\shipout\box\mmz@box
1118 <context> \mmz@temp
1119         \endgroup

```

Advance the counter of shipped-out externs. We do this before preparing the recording information below, because the extern extraction tools expect the extern page numbering to start with 1.

```
1120 \global\advance\mmzExternPages1
```

Prepare the macros which may be used in `record/<type>/new extern` code.

```
1121 \edef\externbasepath{\mmz@extern@basepath}%
```

Adding up the counters below should result in the real page number of the extern. Macro `\mmzRegularPages` holds the number of pages which were shipped out so far using the regular shipout routine of the format; `\mmzExternPages` holds the number of shipped-out extern pages; and `\mmzExtraPages` holds, or at least should hold, the number of pages shipped out using any other means.

```

1122 \edef\pagenumber{%
1123     \the\numexpr\mmzRegularPages

```

In L^AT_EX, the `\mmzRegularPages` holds to number of pages already shipped out. In ConT_EXt, the counter is already increased while processing the page, so we need to subtract 1.

```

1124 <context>     -1%
1125             +\mmzExternPages+\mmzExtraPages
1126             }%

```

Record the creation of the new extern. We do this after shipping out the extern page, so that the recording mechanism can serve as an after-shipout hook, for the unlikely situation that some package really needs to do something when our shipout happens. Note that we absolutely refuse to provide a before-shipout hook, because we can't allow anyone messing with our extern, and that using this after-shipout “hook” is unnecessary for counting extern shipouts, as we already provide this information in the public counter `\mmzExternPages`.

```
1127 \mmzset{record/new extern/.expanded=\mmz@extern@path}%
```

Advance the sequential number of the extern, in the context of the current memoized code chunk. This extern numbering starts at 0, so we only do this after we wrote the cc-memo and called `record/new extern`.

```

1128 \global\advance\mmz@seq1
1129 }

```

`\mmz@shipout@mag` This macro applies the inverse magnification, so that the extern ends up with its natural size on the extern page.

```
1130 \def\mmz@shipout@mag{%
```

We scale the extern box using the PDF primitives: `q` and `Q` save and restore the current graphics state; `cm` applies the given coordinate transformation matrix. ($a\ b\ c\ d\ e\ f$ `cm` transforms (x, y) into $(ax + cy + e, bx + dy + f)$.)

```

1131 \setbox\mmz@box\hbox{%
1132     \pdfliteral{q \mmz@inverse@mag\space 0 0 \mmz@inverse@mag\space 0 0 cm}%
1133     \copy\mmz@box\relax
1134     \pdfliteral{Q}%
1135 }%

```

We first have to scale the paddings, as they might refer to the `\width` etc. of the extern.

```

1136 \dimen0=\dimexpr\mmz@padding@left\relax
1137 \edef\mmz@padding@left{\the\dimexpr\mmz@inverse@mag\dimen0}%
1138 \dimen0=\dimexpr\mmz@padding@bottom\relax
1139 \edef\mmz@padding@bottom{\the\dimexpr\mmz@inverse@mag\dimen0}%
1140 \dimen0=\dimexpr\mmz@padding@right\relax
1141 \edef\mmz@padding@right{\the\dimexpr\mmz@inverse@mag\dimen0}%
1142 \dimen0=\dimexpr\mmz@padding@top\relax
1143 \edef\mmz@padding@top{\the\dimexpr\mmz@inverse@mag\dimen0}%

```

Scale the extern box.

```

1144 \wd\mmz@box=\mmz@inverse@mag\wd\mmz@box\relax
1145 \ht\mmz@box=\mmz@inverse@mag\ht\mmz@box\relax
1146 \dp\mmz@box=\mmz@inverse@mag\dp\mmz@box\relax
1147 }

```

`\mmz@inverse@mag` The inverse magnification factor, i.e. the number we have to multiply the extern dimensions with so that they will end up in their natural size. We compute it, once and for all, at the beginning of the document. To do that, we borrow the little macro `\Pgfg@geT` from `pgfutil-common` (but rename it).

```

1148 {\catcode`\p=12\catcode`\t=12\gdef\mmz@Pgfg@geT#1pt{#1}}
1149 \mmzset{begindocument/.append code={%
1150 \edef\mmz@inverse@mag{\expandafter\mmz@Pgfg@geT\the\dimexpr 1000pt/\mag}%
1151 }}

```

`\mmzRegularPages` This counter holds the number of pages shipped out by the format's shipout routine. \LaTeX and \ConTeXt keep track of this in dedicated counters, so we simply use those. In plain \TeX , we have to hack the `\shipout` macro to install our own counter. In fact, we already did this while loading the required packages, in order to avoid it being redefined by `atbegshi` first. All that is left to do here is to declare the counter.

```

1152 <latex>\let\mmzRegularPages\ReadonlyShipoutCounter
1153 <context>\let\mmzRegularPages\realpageno
1154 <plain>\newcount\mmzRegularPages

```

`\mmzExternPages` This counter holds the number of extern pages shipped out so far.

```

1155 \newcount\mmzExternPages

```

The total number of new externs is announced at the end of the compilation, so that \TeX editors, `latexmk` and such can propose recompilation.

```

1156 \mmzset{
1157   enddocument/afterlastpage/.append code={%
1158     \ifnum\mmzExternPages>0
1159       \PackageWarning{memoize}{The compilation produced \the\mmzExternPages\space
1160         new extern\ifnum\mmzExternPages>1 s\fi}%
1161     \fi
1162   },
1163 }

```

`\mmzExtraPages` This counter will probably remain at zero forever. It should be advanced by any package which (like `Memoize`) ships out pages bypassing the regular shipout routine of the format.

```

1164 \newcount\mmzExtraPages

```

`\mmz@include@extern` This macro, called from cc-memos as `\mmzIncludeExtern`, inserts an extern file into the document. #1 is the sequential number, #2 is either `\hbox` or `\vbox`, #3, #4 and #5 are the (expected) width, height and the depth of the externalized box; #6–#9 are the paddings (left, bottom, right, and top).

```
1165 \def\mmz@include@extern#1#2#3#4#5#6#7#8#9{%
```

Set the extern sequential number, so that we open the correct extern file (`\mmz@extern@basename`).

```
1166 \mmz@seq=#1\relax
```

Use the primitive PDF graphics inclusion commands to include the extern file. Set the correct depth or the resulting box, and shift it as specified by the padding.

```
1167 \setbox\mmz@box=#2{%
1168   \setbox0=\hbox{%
1169     \lower\dimexpr #5+#7\relax\hbox{%
1170       \hskip -#6\relax
1171       \setbox0=\hbox{%
1172         \mmz@insertpdfpage{\mmz@extern@path}{1}%
1173       }%
1174       \unhbox0
1175     }%
1176   }%
1177   \wd0 \dimexpr\wd0-#8\relax
1178   \ht0 \dimexpr\ht0-#9\relax
1179   \dp0 #5\relax
1180   \box0
1181 }%
```

Check whether the size of the included extern is as expected. There is no need to check `\dp`, we have just set it. (`\mmz@if@roughly@equal` is defined in section 4.3.)

```
1182 \mmz@tempfalse
1183 \mmz@if@roughly@equal{\mmz@tolerance}{#3}{\wd\mmz@box}{%
1184   \mmz@if@roughly@equal{\mmz@tolerance}{#4}{\ht\mmz@box}{%
1185     \mmz@temptrue
1186   }{}{}%
1187 \ifmmz@temp
1188 \else
1189   \mmz@use@memo@warning{\mmz@extern@path}{#3}{#4}{#5}%
1190 \fi
```

Use the extern box, with the precise size as remembered at memoization.

```
1191 \wd\mmz@box=#3\relax
1192 \ht\mmz@box=#4\relax
1193 \box\mmz@box
```

Record that we have used this extern.

```
1194 \pgfkeysalso{/mmz/record/used extern={\mmz@extern@path}}%
1195 }
```

```
1196 \def\mmz@use@memo@warning#1#2#3#4{%
1197   \PackageWarning{memoize}{Unexpected size of extern "#1";
1198     expected #2\space x \the\dimexpr #3+#4\relax,
1199     got \the\wd\mmz@box\space x \the\dimexpr\the\ht\mmz@box+\the\dp\mmz@box\relax}%
1200 }
```

`\mmz@insertpdfpage` This macro inserts a page from the PDF into the document. We define it according to which engine is being used. Note that ConT_EXt always uses LuaT_EX.

```
1201 <latex,plain>\ifdef\luatexversion{%
```

```

1202 \def\mmz@insertpdfpage#1#2{% #1 = filename, #2 = page number
1203   \saveimageresource page #2 mediabox {#1}%
1204   \useimageresource\lastsavedimageresourceindex
1205 }%
1206 <*\latex,plain>
1207 }{%
1208 \ifdef\XeTeXversion{%
1209   \def\mmz@insertpdfpage#1#2{%
1210     \XeTeXpdfpage #1 page #2 media
1211   }%
1212 }{% pdfLaTeX
1213   \def\mmz@insertpdfpage#1#2{%
1214     \pdfximage page #2 mediabox {#1}%
1215     \pdfrefximage\pdflastximage
1216   }%
1217 }%
1218 }
1219 </\latex,plain>

```

`\mmz@include@extern@from@tbe@box` Include the extern number #1 residing in `\mmz@tbe@box` into the document.

It may be called as `\mmzIncludeExtern` from after memoization hook if `\ifmmzkeepexterns` was set to true during memoization. The macro takes the same arguments as `\mmzIncludeExtern` but disregards all but the first one, the extern sequential number. Using this macro, a complex memoization driver can process the cc-memo right after memoization, by issuing `\global\mmzkeepexternstrue\xtoksapp\mmzAfterMemoizationExtra{\the\mmzCCMemo}`.

```

1220 \def\mmz@include@extern@from@tbe@box#1#2#3#4#5#6#7#8#9{%
1221   \setbox0\vbox{%
1222     \@tempcnta#1\relax
1223     \vskip1pt
1224     \unvcopy\mmz@tbe@box
1225     \@whilenum\@tempcnta>0\do{%
1226       \setbox0\lastbox
1227       \advance\@tempcnta-1\relax
1228     }%
1229     \global\setbox1\lastbox
1230     \@whiles\ifdim0pt=\lastskip\fi{%
1231       \setbox0\lastbox
1232     }%
1233     \box\mmz@box
1234   }%
1235   \box1
1236 }

```

4 Extraction

4.1 Extraction mode and method

extract This key selects the extraction mode and method. It normally occurs in the package options list, less commonly in the preamble, and never in the document body.

```

1237 \def\mmzvalueof#1{\pgfkeysvalueof{/mmz/#1}}
1238 \mmzset{
1239   extract/.estore in=\mmz@extraction@method,
1240   extract/.value required,
1241   begindocument/.append style={extract/.code=\mmz@preamble@only@error},

```

extract/perl Any other value will select internal extraction with the given method. Memoize ships with two extraction scripts, a Perl script and a Python script, which are selected by `extract=perl` (the default) and `extract=python`, respectively. We run the scripts in verbose mode (without `-q`), and keep the `.mmz` file as is (without `-k`), i.e. we're not commenting out the `\mmzNewExtern`

lines, because we're about to overwrite it anyway. We inform the script about the format of the document (-F).

```

1242 extract/perl/.code={%
1243   \mmz@clear@extraction@log
1244   \pdf@system{%
1245     \mmzvalueof{perl extraction command}\space
1246     \mmzvalueof{perl extraction options}%
1247   }%
1248   \mmz@check@extraction@log{perl}%
1249   \def\mmz@mknkdir@command{\mmzvalueof{perl extraction command} --mkdir}%
1250 },
1251 perl extraction command/.initial=memoize-extract.pl,
1252 perl extraction options/.initial={\space
1253 <latex> -F latex
1254 <plain> -F plain
1255 <context> -F context
1256   \jobname\space
1257 },
1258 extract=perl,
1259 extract/python/.code={%
1260   \mmz@clear@extraction@log
1261   \pdf@system{%
1262     \mmzvalueof{python extraction command}\space
1263     \mmzvalueof{python extraction options}%
1264   }%
1265   \mmz@check@extraction@log{python}%
1266   \def\mmz@mknkdir@command{\mmzvalueof{python extraction command} --mkdir}%
1267 },
1268 python extraction command/.initial=memoize-extract.py,
1269 python extraction options/.initial={\space
1270 <latex> -F latex
1271 <plain> -F plain
1272 <context> -F context
1273   \jobname\space
1274 },
1275 }
1276 \def\mmz@preamble@only@error{%
1277   \PackageError{memoize}{%
1278     Ignoring the invocation of "\pgfkeyscurrentkey".
1279     This key may only be executed in the preamble}{}%
1280 }

```

The extraction log — As we cannot access the exit status of a system command in T_EX, we communicate with the system command via the “extraction log file,” produced by both T_EX-based extraction and the Perl and Python extraction script. This file signals whether the embedded extraction was successful — if it is, the file ends if `\endinput` — and also contains any warnings and errors thrown by the script. As the log is really a T_EX file, the idea is to simply input it after extracting each extern (for T_EX-based extraction) or after the extraction of all externs (for the external scripts).

```

1281 \def\mmz@clear@extraction@log{%
1282   \begingroup
1283   \immediate\openout0{\jobname.mmz.log}%
1284   \immediate\closeout0
1285   \endgroup
1286 }

```

#1 is the extraction method.

```

1287 \def\mmz@check@extraction@log#1{%
1288   \begingroup \def\extractionmethod{#1}%
1289   \mmz@tempfalse \let\mmz@orig@endinput\endinput

```



```

1290 \def\endinput{\mmz@temptrue\mmz@orig@endinput}%
1291 \@input{\jobname.mmz.log}%
1292 \ifmmz@temp \else \mmz@extraction@error \fi \endgroup }
1293 \def\mmz@extraction@error{%
1294 \PackageError{memoize}{Extraction of externs from document
1295 "\jobname.pdf" using method "\extractionmethod" was
1296 unsuccessful}{The extraction script "\mmzvalueof{\extractionmethod\space
1297 extraction command}" wasn't executed or didn't finish execution
1298 properly.}}

```

4.2 The record files

record This key activates a record *<type>*: the hooks defined by that record *<type>* will henceforth be executed at the appropriate places.

A *<hook>* of a particular *<type>* resides in `pgfkeys` path `/mmz/record/<type>/<hook>`, and is invoked via `/mmz/record/<hook>`. Record type activation thus appends a call of the former to the latter. It does so using handler `.try`, so that unneeded hooks may be left undefined.

```

1299 \mmzset{
1300   record/.style={%
1301     record/begin/.append style={
1302       /mmz/record/#1/begin/.try,

```

The `begin` hook also executes the `prefix` hook, so that `\mmzPrefix` surely occurs at the top of the `.mmz` file. Listing each prefix type separately in this hook ensures that `prefix` of a certain type is executed after that type's `begin`.

```

1303     /mmz/record/#1/prefix/.try/.expanded=\mmz@prefix,
1304   },
1305   record/prefix/.append style={/mmz/record/#1/prefix/.try={##1}},
1306   record/new extern/.append style={/mmz/record/#1/new extern/.try={##1}},
1307   record/used extern/.append style={/mmz/record/#1/used extern/.try={##1}},
1308   record/new cmemo/.append style={/mmz/record/#1/new cmemo/.try={##1}},
1309   record/new ccmemo/.append style={/mmz/record/#1/new ccmemo/.try={##1}},
1310   record/used cmemo/.append style={/mmz/record/#1/used cmemo/.try={##1}},
1311   record/used ccmemo/.append style={/mmz/record/#1/used ccmemo/.try={##1}},
1312   record/end/.append style={/mmz/record/#1/end/.try},
1313 },
1314 }

```

no record This key deactivates all record types. Below, we use it to initialize the relevant keys; in the user code, it may be used to deactivate the preactivated `mmz` record type.

```

1315 \mmzset{
1316   no record/.style={%

```

The `begin` hook clears itself after invocation, to prevent double execution. Consequently, `record/begin` may be executed by the user in the preamble, without any ill effects.

```

1317   record/begin/.style={record/begin/.style={}},

```

The `prefix` key invokes itself again when the group closes. This way, we can correctly track the path prefix changes in the `.mmz` even if `path` is executed in a group.

```

1318   record/prefix/.code={\aftergroup\mmz@record@prefix},
1319   record/new extern/.code={},
1320   record/used extern/.code={},
1321   record/new cmemo/.code={},
1322   record/new ccmemo/.code={},
1323   record/used cmemo/.code={},
1324   record/used ccmemo/.code={},

```


The `end` hook clears itself after invocation, to prevent double execution. Consequently, `record/end` may be executed by the user before the end of the document, without any ill effects.

```
1325   record/end/.style={record/end/.code={}},
1326 }
1327 }
```

We define this macro because `\aftergroup`, used in `record/prefix`, only accepts a token.

```
1328 \def\mmz@record@prefix{%
1329   \mmzset{/mmz/record/prefix/.expanded=\mmz@prefix}%
1330 }
```

Initialize the hook keys, preactivate `mmz` record type, and execute hooks `begin` and `end` at the edges of the document.

```
1331 \mmzset{
1332   no record,
1333   record=mmz,
1334   begindocument/.append style={record/begin},
1335   enddocument/afterlastpage/.append style={record/end},
1336 }
```

4.2.1 The .mmz file

Think of the `.mmz` record file as a \TeX -readable log file, which lets the extraction procedure know what happened in the previous compilation. The file is in \TeX format, so that we can trigger internal \TeX -based extraction by simply inputting it. The commands it contains are intentionally as simple as possible (just a macro plus braced arguments), to facilitate parsing by the external scripts.

`record/mmz/...` These hooks simply put the calls of the corresponding macros into the file. All but hooks but `begin` and `end` receive the full path to the relevant file as the only argument (ok, `prefix` receives the full path prefix, as set by key `path`).

```
1337 \mmzset{
1338   record/mmz/begin/.code={%
1339     \newwrite\mmz@mmzout
```

The record file has a fixed name (the jobname plus the `.mmz` suffix) and location (the current directory, i.e. the directory where \TeX is executed from; usually, this will be the directory containing the \TeX source).

```
1340   \immediate\openout\mmz@mmzout{\jobname.mmz}%
1341 },
```

The `\mmzPrefix` is used by the clean-up script, which will remove all files with the given path prefix but (unless called with `--all`) those mentioned in the `.mmz`. Now this script could in principle figure out what to remove by inspecting the paths to utilized/created memos/externs in the `.mmz` file, but this method could lead to problems in case of an incomplete (perhaps empty) `.mmz` file created by a failed compilation. Recording the path prefix in the `.mmz` radically increases the chances of a successful clean-up, which is doubly important, because a clean-up is sometimes precisely what we need to do to recover after a failed compilation.

```
1342   record/mmz/prefix/.code={%
1343     \immediate\write\mmz@mmzout{\noexpand\mmzPrefix{#1}}}%
1344 },
1345   record/mmz/new extern/.code={%
```

While this key receives a single formal argument, Memoize also prepares macros `\externbasepath` (`#1` without the `.pdf` suffix), `\pagenumber` (of the extern page in the document PDF), and `\expectedwidth` and `\expectedheight` (of the extern page).

```
1346 \immediate\write\mmz@mmzout{%
1347 \noexpand\mmzNewExtern{#1}{\pagenumber}{\expectedwidth}{\expectedheight}%
1348 }%
```

Support latexmk:

```
1349 <latex> \typeout{No file #1}%
1350 },
1351 record/mmz/new cmemo/.code={%
1352 \immediate\write\mmz@mmzout{\noexpand\mmzNewCMemo{#1}}%
1353 },
1354 record/mmz/new ccmemo/.code={%
1355 \immediate\write\mmz@mmzout{\noexpand\mmzNewCCMemo{#1}}%
1356 },
1357 record/mmz/used extern/.code={%
1358 \immediate\write\mmz@mmzout{\noexpand\mmzUsedExtern{#1}}%
1359 },
1360 record/mmz/used cmemo/.code={%
1361 \immediate\write\mmz@mmzout{\noexpand\mmzUsedCMemo{#1}}%
1362 },
1363 record/mmz/used ccmemo/.code={%
1364 \immediate\write\mmz@mmzout{\noexpand\mmzUsedCCMemo{#1}}%
1365 },
1366 record/mmz/end/.code={%
```

Add the `\endinput` marker to signal that the file is complete.

```
1367 \immediate\write\mmz@mmzout{\noexpand\endinput}%
1368 \immediate\closeout\mmz@mmzout
1369 },
```

4.2.2 The shell scripts

We define two shell script record types: `sh` for Linux, and `bat` for Windows.

`sh` These keys set the shell script filenames.

`bat`

```
1370 sh/.store in=\mmz@shname,
1371 sh=memoize-extract.\jobname.sh,
1372 bat/.store in=\mmz@batname,
1373 bat=memoize-extract.\jobname.bat,
```

`record/sh/...` Define the Linux shell script record type.

```
1374 record/sh/begin/.code={%
1375 \newwrite\mmz@shout
1376 \immediate\openout\mmz@shout{\mmz@shname}%
1377 },
1378 record/sh/new extern/.code={%
1379 \begingroup
```

Macro `\mmz@tex@extraction@systemcall` is customizable through `tex extraction command`, `tex extraction options` and `tex extraction script`.

```
1380 \immediate\write\mmz@shout{\mmz@tex@extraction@systemcall}%
1381 \endgroup
1382 },
1383 record/sh/end/.code={%
1384 \immediate\closeout\mmz@shout
1385 },
```

`record/bat/...` Rinse and repeat for Windows.

```
1386 record/bat/begin/.code={%
1387   \newwrite\mmz@batout
1388   \immediate\openout\mmz@batout{\mmz@batname}%
1389 },
1390 record/bat/new extern/.code={%
1391   \begingroup
1392   \immediate\write\mmz@batout{\mmz@tex@extraction@systemcall}%
1393   \endgroup
1394 },
1395 record/bat/end/.code={%
1396   \immediate\closeout\mmz@batout
1397 },
```

4.2.3 The Makefile

The implementation of the Makefile record type is the most complex so far, as we need to keep track of the targets.

`makefile` This key sets the makefile filename.

```
1398 makefile/.store in=\mmz@makefilename,
1399 makefile=memoize-extract.\jobname.makefile,
1400 }
```

We need to define a macro which expands to the tab character of catcode “other”, to use as the recipe prefix.

```
1401 \begingroup
1402 \catcode\^^I=12
1403 \gdef\mmz@makefile@recipe@prefix{^^I}%
1404 \endgroup
```

`record/makefile/...` Define the Makefile record type.

```
1405 \mmzset{
1406   record/makefile/begin/.code={%
```

We initialize the record type by opening the file and setting makefile variables `.DEFAULT_GOAL` and `.PHONY`.

```
1407   \newwrite\mmz@makefileout
1408   \newtoks\mmz@makefile@externs
1409   \immediate\openout\mmz@makefileout{\mmz@makefilename}%
1410   \immediate\write\mmz@makefileout{.DEFAULT_GOAL = externs}%
1411   \immediate\write\mmz@makefileout{.PHONY: externs}%
1412 },
```

The crucial part, writing out the extraction rule. The target comes first, then the recipe, which is whatever the user has set by `tex extraction command`, `tex extraction options` and `tex extraction script`.

```
1413   record/makefile/new extern/.code={%
```

The target extern file:

```
1414   \immediate\write\mmz@makefileout{#1:}%
1415   \begingroup
```

The recipe is whatever the user set by `tex extraction command`, `tex extraction options` and `tex extraction script`.

```
1416   \immediate\write\mmz@makefileout{%
1417     \mmz@makefile@recipe@prefix\mmz@tex@extraction@systemcall}%
1418   \endgroup
```

Append the extern file to list of targets.

```

1419 \xtoksapp\mmz@makefile@externs{#1\space}%
1420 },
1421 record/makefile/end/.code={%

```

Before closing the file, we list the extern files as the prerequisites of our phony default target, `externs`.

```

1422 \immediate\write\mmz@makefileout{externs: \the\mmz@makefile@externs}%
1423 \immediate\closeout\mmz@makefileout
1424 },
1425 }

```

4.3 T_EX-based extraction

`extract/tex` We trigger the T_EX-based extraction by inputting the `.mmz` record file.

```

1426 \mmzset{
1427   extract/tex/.code={%
1428     \begingroup
1429     \@input{\jobname.mmz}%
1430     \endgroup
1431   },
1432 }

```

`\mmzUsedCMemo` We can ignore everything but `\mmzNewExterns`. All these macros receive a single argument.

```

\mmzUsedCCMemo 1433 \def\mmzUsedCMemo#1{}
\mmzUsedExtern 1434 \def\mmzUsedCCMemo#1{}
\mmzNewCMemo    1435 \def\mmzUsedExtern#1{}
\mmzNewCCMemo   1436 \def\mmzNewCMemo#1{}
\mmzPrefix     1437 \def\mmzNewCCMemo#1{}
               1438 \def\mmzPrefix#1{}

```

`\mmzNewExtern` Command `\mmzNewExtern` takes four arguments. It instructs us to extract page #2 of document `\jobname.pdf` to file #1. During the extraction, we will check whether the size of the extern matches the given expected width (#3) and total height (#4).

We perform the extraction by an embedded T_EX call. The system command that gets executed is stored in `\mmz@tex@extraction@systemcall`, which is set by `tex extraction command` and friends; by default, we execute `pdftex`.

```

1439 \def\mmzNewExtern#1{%

```

The T_EX executable expects the basename as the argument, so we strip away the `.pdf` suffix.

```

1440 \mmz@new@extern@i#1\mmz@temp
1441 }
1442 \def\mmz@new@extern@i#1.pdf\mmz@temp#2#3#4{%
1443   \begingroup

```

Define the macros used in `\mmz@tex@extraction@systemcall`.

```

1444 \def\externbasepath{#1}%
1445 \def\pagenumber{#2}%
1446 \def\expectedwidth{#3}%
1447 \def\expectedheight{#4}%

```

Empty out the extraction log.

```

1448 \mmz@clear@extraction@log

```

Extract.

```

1449 \pdf@system{\mmz@tex@extraction@systemcall}%

```

Was the extraction successful? We temporarily redefine the extraction error message macro (suited for the external extraction scripts, which extract all externs in one go) to report the exact problematic extern page.

```

1450 \let\mmz@extraction@error\mmz@pageextraction@error
1451 \mmz@check@extraction@log{tex}%
1452 \endgroup
1453 }

1454 \def\mmz@pageextraction@error{%
1455 \PackageError{memoize}{Extraction of extern page \pagenumber\space from
1456 document "jobname.pdf" using method "\extractionmethod" was
1457 unsuccessful.}{Check the log file to see if the extraction script was
1458 executed at all, and if it finished successfully. You might also want to
1459 inspect "\externbasepath.log", the log file of the embedded TeX compilation
1460 which ran the extraction script}}

```

tex extraction command Using these keys, we set the system call which will be invoked for each extern page. The **tex extraction options** value of this key is expanded when executing the system command. The user may deploy **tex extraction script** the following macros in the value of these keys:

- `\externbasepath`: the extern PDF that should be produced, minus the `.pdf` suffix;
- `\pagenumber`: the page number to be extracted;
- `\expectedwidth`: the expected width of the extracted page;
- `\expectedheight`: the expected total height of the extracted page;

```

1461 \def\mmz@tex@extraction@systemcall{%
1462 \mmzvalueof{tex extraction command}\space
1463 \mmzvalueof{tex extraction options}\space
1464 "\mmzvalueof{tex extraction script}"%
1465 }

```

The default system call for T_EX-based extern extraction. As this method, despite being T_EX-based, shares no code with the document, we're free to implement it with any engine and format we want. For reasons of speed, we clearly go for the plain pdfT_EX.³ We perform the extraction by a little T_EX script, `memoize-extract-one`, inputted at the end of the value given to `tex extraction script`.

```

1466 \mmzset{
1467 tex extraction command/.initial=pdftex,
1468 tex extraction options/.initial={%
1469 -halt-on-error
1470 -interaction=batchmode
1471 -jobname "\externbasepath"
1472 },
1473 tex extraction script/.initial={%
1474 \def\noexpand\fromdocument{\jobname.pdf}%
1475 \def\noexpand\pagenumber{\pagenumber}%
1476 \def\noexpand\expectedwidth{\expectedwidth}%
1477 \def\noexpand\expectedheight{\expectedheight}%
1478 \def\noexpand\logfile{\jobname.mmz.log}%
1479 \unexpanded{%
1480 \def\warningtemplate{%
1481 \noexpand\PackageWarning{memoize}{\warningtext}%
1482 \plain\warning{memoize: \warningtext}%
1483 \context\warning{memoize: \warningtext}%
1484 }}%
1485 \ifdef\XeTeXversion{}{%
1486 \def\noexpand\mmzpdfmajorversion{\the\pdfmajorversion}%
1487 \def\noexpand\mmzpdfminorversion{\the\pdfminorversion}%
1488 }%
1489 \noexpand\input memoize-extract-one

```

³I implemented the first version of T_EX-based extraction using L^AT_EX and package `graphicx`, and it was (running with pdfT_EX engine) almost four times slower than the current plain T_EX implementation.

```

1490 },
1491 }
1492 </mmz>

```

4.3.1 memoize-extract-one.tex

The rest of the code of this section resides in file `memoize-extract-one.tex`. It is used to extract a single extern page from the document; it also checks whether the extern page dimensions are as expected, and passes a warning to the main job if that is not the case. For the reason of speed, the extraction script is in plain \TeX format. For the same reason, it is compiled by `pdf \TeX` engine by default, but we nevertheless take care that it will work with other (supported) engines as well.

```

1493 <*extract-one>
1494 \catcode`\@11\relax
1495 \def\@firstoftwo#1#2{#1}
1496 \def\@secondoftwo#1#2{#2}

```

Set the PDF version (maybe) passed to the script via `\mmzpdfmajorversion` and `\mmzpdfminorversion`.

```

1497 \ifdefined\XeTeXversion
1498 \else
1499   \ifdefined\luatexversion
1500     \def\pdfmajorversion{\pdfvariable majorversion}%
1501     \def\pdfminorversion{\pdfvariable minorversion}%
1502   \fi
1503   \ifdefined\mmzpdfmajorversion
1504     \pdfmajorversion\mmzpdfmajorversion\relax
1505   \fi
1506   \ifdefined\mmzpdfminorversion
1507     \pdfminorversion\mmzpdfminorversion\relax
1508   \fi
1509 \fi

```

Allocate a new output stream, always — `\newwrite` is `\outer` and thus cannot appear in a conditional.

```

1510 \newwrite\extractionlog

```

Are we requested to produce a log file?

```

1511 \ifdefined\logfile
1512   \immediate\openout\extractionlog{\logfile}%

```

Define a macro which both outputs the warning message and writes it to the extraction log.

```

1513   \def\doublewarning#1{%
1514     \message{#1}%
1515     \def\warningtext{#1}%

```

This script will be called from different formats, so it is up to the main job to tell us, by defining macro `\warningtemplate`, how to throw a warning in the log file.

```

1516     \immediate\write\extractionlog{%
1517       \ifdefined\warningtemplate\warningtemplate\else\warningtext\fi
1518     }%
1519   }%
1520 \else
1521   \let\doublewarning\message
1522 \fi
1523 \newif\ifforce
1524 \ifdefined\force
1525   \csname force\force\endcsname
1526 \fi

```

`\mmz@if@roughly@equal` This macro checks whether the given dimensions (#2 and #3) are equal within the tolerance given by #1. We use the macro both in the extraction script and in the main package. (We don't use `\ifpdfabsdim`, because it is unavailable in X_YTeX.)

```

1527 \extract-one)
1528 *mmz,extract-one)
1529 \def\mmz@tolerance{0.01pt}
1530 \def\mmz@if@roughly@equal#1#2#3{%
1531   \dimen0=\dimexpr#2-#3\relax
1532   \ifdim\dimen0<0pt
1533     \dimen0=-\dimen0\relax
1534   \fi
1535   \ifdim\dimen0>#1\relax
1536     \expandafter\@secondoftwo
1537   \else

```

The exact tolerated difference is, well, tolerated. This is a must to support `tolerance=0pt`.

```

1538   \expandafter\@firstoftwo
1539   \fi
1540 }%
1541 \mmz,extract-one)
1542 *extract-one)

```

Grab the extern page from the document and put it in a box.

```

1543 \ifdefined\XeTeXversion
1544   \setbox0=\hbox{\XeTeXpdffile \fromdocument\space page \pagenumber media}%
1545 \else
1546   \ifdefined\luatexversion
1547     \saveimageresource page \pagenumber mediabox {\fromdocument}%
1548     \setbox0=\hbox{\useimageresource\lastsavedimageresourceindex}%
1549   \else
1550     \pdfximage page \pagenumber mediabox {\fromdocument}%
1551     \setbox0=\hbox{\pdfrefximage\pdflastximage}%
1552   \fi
1553 \fi

```

Check whether the extern page is of the expected size.

```

1554 \newif\ifbaddimensions
1555 \ifdefined\expectedwidth
1556   \ifdefined\expectedheight
1557     \mmz@if@roughly@equal{\mmz@tolerance}{\wd0}{\expectedwidth}{%
1558       \mmz@if@roughly@equal{\mmz@tolerance}{\ht0}{\expectedheight}%
1559     }%
1560     {\baddimensionstrue}%
1561   }{\baddimensionstrue}%
1562 \fi
1563 \fi

```

We'll setup the page geometry of the extern file and shipout the extern — if all is well, or we're forced to do it.

```

1564 \ifdefined\luatexversion
1565   \let\pdfpagewidth\pagewidth
1566   \let\pdfpageheight\pageheight
1567   \def\pdfhorigin{\pdfvariable horigin}%
1568   \def\pdfvorigin{\pdfvariable vorigin}%
1569 \fi
1570 \def\do@shipout{%
1571   \pdfpagewidth=\wd0
1572   \pdfpageheight=\ht0
1573   \ifdefined\XeTeXversion

```

```

1574 \hoffset -1 true in
1575 \voffset -1 true in
1576 \else
1577 \pdfhorigin=0pt
1578 \pdfvorigin=0pt
1579 \fi
1580 \shipout\box0
1581 }
1582 \ifbaddimensions
1583 \doublewarning{I refuse to extract page \pagenumber\space from
1584 "\fromdocument", because its size (\the\wd0 \space x \the\ht0) is not
1585 what I expected (\expectedwidth\space x \expectedheight)}%
1586 \ifforce\do@shipout\fi
1587 \else
1588 \do@shipout
1589 \fi

```

If logging is in effect and the extern dimensions were not what we expected, write a warning into the log.

```

1590 \ifdefined\logfile
1591 \immediate\write\extractionlog{\noexpand\endinput}%
1592 \immediate\closeout\extractionlog
1593 \fi
1594 \bye
1595 </extract-one>

```

5 Automemoization

Install the advising framework implemented by our auxiliary package Advice, which automemoization depends on. This will define keys `auto`, `activate` etc. in our keypath.

```

1596 <*mmz>
1597 \mmzset{
1598 .install advice={setup key=auto, activation=deferred},

```

We switch to the immediate activation at the end of the preamble.

```

1599 begindocument/before/.append style={activation=immediate},
1600 }

```

manual Unless the user switched on `manual`, we perform the deferred (de)activations at the beginning of the document (and then clear the style, so that any further deferred activations will start with a clean slate). In \LaTeX , we will use the latest possible hook, `begindocument/end`, as we want to hack into commands defined by other packages. (The \TeX conditional needs to be defined before using it in `.append` code below.

```

1601 \newif\ifmmz@manual
1602 \mmzset{
1603 manual/.is if=mmz@manual,
1604 begindocument/end/.append code={%
1605 \ifmmz@manual
1606 \else
1607 \pgfkeysalso{activate deferred,activate deferred/.code={}}%
1608 \fi
1609 },

```

Announce Memoize's run conditions and handlers.

```

1610 auto/.cd,
1611 run if memoization is possible/.style={
1612 run conditions=\mmz@auto@rc@if@memoization@possible

```



```

1613 },
1614 run if memoizing/.style={run conditions=\mmz@auto@rc@if@memoizing},
1615 apply options/.style={
1616     bailout handler=\mmz@auto@bailout,
1617     outer handler=\mmz@auto@outer,
1618 },
1619 memoize/.style={
1620     run if memoization is possible,
1621     apply options,
1622     inner handler=\mmz@auto@memoize
1623 },
1624 <*/latex>
1625 noop/.style={run if memoization is possible, noop \AdviceType},
1626 noop command/.style={apply options, inner handler=\mmz@auto@noop},
1627 noop environment/.style={
1628     outer handler=\mmz@auto@noop@env, bailout handler=\mmz@auto@bailout},
1629 </latex>
1630 <plain, context> noop/.style={inner handler=\mmz@auto@noop},
1631 nomemoize/.style={noop, options=disable},
1632 replicate/.style={run if memoizing, inner handler=\mmz@auto@replicate},
1633 to context/.style={run if memoizing, outer handler=\mmz@auto@tocontext},
1634 }

```

Abortion We cheat and let the `run conditions` do the work — it is cheaper to just always abort than to invoke the outer handler. (As we don't set `\AdviceRuntrue`, the run conditions will never be satisfied.)

```

1635 \mmzset{
1636     auto/abort/.style={run conditions=\mmzAbort},
1637 }

```

And the same for unmemoizable:

```

1638 \mmzset{
1639     auto/unmemoizable/.style={run conditions=\mmzUnmemoizable},
1640 }

```

For one, we abort upon `\pdfsavepos` (called `\savepos` in LuaTeX). Second, unless in LuaTeX, we submit `\errmessage`, which allows us to detect at least some errors — in LuaTeX, we have a more bullet-proof system of detecting errors, see `\mmz@memoize` in §3.2.

```

1641 \ifdef\luatexversion{%
1642     \mmzset{auto=\savepos{abort}}
1643 }{%
1644     \mmzset{
1645         auto=\pdfsavepos{abort},
1646         auto=\errmessage{abort},
1647     }
1648 }

```

run if memoization is possible These run conditions are used by `memoize` and `noop`: Memoize should be `\mmz@auto@rc@if@memoization@possible` enabled, but we should not be already within Memoize, i.e. memoizing or normally compiling some code submitted to memoization.

```

1649 \def\mmz@auto@rc@if@memoization@possible{%
1650     \ifmemoize
1651         \ifinmemoize
1652         \else
1653             \AdviceRuntrue
1654         \fi
1655     \fi
1656 }

```

`run if memoizing` These run conditions are used by `\label` and `\ref`: they should be handled only during `\mmz@auto@rc@if@memoizing` memoization (which implies that Memoize is enabled).

```
1657 \def\mmz@auto@rc@if@memoizing{%
1658   \ifmemoizing\AdviceRuntrue\fi
1659 }
```

`\mmznext` The next-options, set by this macro, will be applied to the next, and only next instance of automemoization. We set the next-options globally, so that only the linear order of the invocation matters. Note that `\mmznext`, being a user command, must also be defined in package `nomemoize`.

```
1660 </mmz>
1661 <nommz> \def\mmznext#1{\ignorespaces}
1662 <*mmz>
1663 \def\mmznext#1{\gdef\mmz@next{#1}\ignorespaces}
1664 \mmznext{}}%
```

`apply options` The outer and the bailout handler defined here work as a team. The outer handler’s job is to `\mmz@auto@outer` apply the auto- and the next-options; therefore, the bailout handler must consume the next- `\mmz@auto@bailout` options as well. To keep the option application local, the outer handler opens a group, which is expected to be closed by the inner handler. This key is used by `memoize` and `noop` command.

```
1665 \def\mmz@auto@outer{%
1666   \begingroup
1667   \mmzAutoInit
1668   \AdviceCollector
1669 }
1670 \def\mmz@auto@bailout{%
1671   \mmznext{}}%
1672 }
```

`\mmzAutoInit` Apply first the auto-options, and then the next-options (and clear the latter). Finally, if we have any extra collector options (set by the `verbatim` keys), append them to Advice’s (raw) collector options.

```
1673 \def\mmzAutoInit{%
1674   \ifdefempty\AdviceOptions{\expandafter\mmzset\expandafter{\AdviceOptions}}%
1675   \ifdefempty\mmz@next{\expandafter\mmzset\expandafter{\mmz@next}\mmznext{}}%
1676   \eappto\AdviceRawCollectorOptions{\expandonce\mmzRawCollectorOptions}%
1677 }
```

`memoize` This key installs the inner handler for memoization. If you compare this handler to the definition `\mmz@auto@memoize` of `\mmz` in section 3.1, you will see that the only thing left to do here is to start memoization with `\Memoize`, everything else is already done by the advising framework, as customized by Memoize.

The first argument to `\Memoize` is the memoization key (which the code `md5sum` is computed off of); it consists of the handled code (the contents of `\AdviceReplaced`) and its arguments, which were collected into `##1`. The second argument is the code which the memoization driver will execute. `\AdviceOriginal`, if invoked right away, would execute the original command; but as this macro is only guaranteed to refer to this command within the advice handlers, we expand it before calling `\Memoize`. that command.

Note that we don’t have to define different handlers for commands and environments, and for different T_EX formats. When memoizing command `\foo`, `\AdviceReplaced` contains `\foo`. When memoizing environment `foo`, `\AdviceReplaced` contains `\begin{foo}`, `\foo` or `\startfoo`, depending on the format, while the closing tag (`\end{foo}`, `\endfoo` or `\stopfoo`) occurs at the end of the collected arguments, because `apply options` appended `\collargsEndTagtrue` to raw collector options.

This macro has no formal parameters, because the collected arguments will be grabbed by `\mmz@marshal`, which we have to go through because executing `\Memoize` closes the memoization

group and we lose the current value of `\ifmmz@ignorespaces`. (We also can't use `\aftergroup`, because closing the group is not the final thing `\Memoize` does.)

```

1678 \long\def\mmz@auto@memoize#1{%
1679   \expanded{%
1680     \noexpand\Memoize
1681     {\expandonce\AdviceReplaced\unexpanded{#1}}%
1682     {\expandonce\AdviceOriginal\unexpanded{#1}}%
1683     \ifmmz@ignorespaces\ignorespaces\fi
1684   }%
1685 }

```

noop The no-operation handler can be used to apply certain options for the span of the execution of the handled command or environment. This is exploited by `auto/nomemoize`, which sets `\mmz@auto@noop@env` `disable` as an auto-option.

The handler for commands and non-L^AT_EX environments is implemented as an inner handler. On its own, it does nothing except honor `verbatim` and `ignore spaces` (only takes care of `verbatim` and `ignore spaces` (in the same way as the memoization handler above), but it is intended to be used alongside the default outer handler, which applies the auto- and the next-options. As that handler opens a group (and this handler closes it), we have effectively delimited the effect of those options to this invocation of the handled command or environment.

```

1686 \long\def\mmz@auto@noop#1{%
1687   \expandafter\mmz@maybe@scantokens\expandafter{\AdviceOriginal#1}%
1688   \expandafter\endgroup
1689   \ifmmz@ignorespaces\ignorespaces\fi
1690 }

```

In L^AT_EX, and only there, commands and environments need separate treatment. As L^AT_EX environments introduce a group of their own, we can simply hook our initialization into the beginning of the environment (as a one-time hook). Consequently, we don't need to collect the environment body, so this can be an outer handler.

```

1691 <*\latex>
1692 \def\mmz@auto@noop@env{%
1693   \AddToHookNext{env/\AdviceName/begin}{%
1694     \mmzAutoInit
1695     \ifmmz@ignorespaces\ignorespacesafterend\fi
1696   }%
1697   \AdviceOriginal
1698 }
1699 </\latex>

```

replicate This inner handler writes a copy of the handled command or environment's invocation into the cc-memo (and then executes it). As it is used alongside `run if memoizing`, the replicated command in the cc-memo will always execute the original command. The system works even if replication is off when the cc-memo is input; in that case, the control sequence in the cc-memo directly executes the original command.

This handler takes an option, `expanded` — if given, the collected arguments will be expanded (under protection) before being written into the cc-memo.

```

1700 \def\mmz@auto@replicate#1{%
1701   \begingroup
1702   \let\mmz@auto@replicate@expansion\unexpanded
1703   \expandafter\pgfqkeys\expandafter{\mmz/auto/replicate}{\AdviceOptions}}%
1704 <\latex> \let\protect\noexpand
1705   \expanded{%
1706     \endgroup
1707     \noexpand\gtoksapp\noexpand\mmzCCMemo{%
1708       \expandonce\AdviceReplaced\mmz@auto@replicate@expansion{#1}}%
1709       \expandonce\AdviceOriginal\unexpanded{#1}%

```

```

1710 }%
1711 }
1712 \pgfqkeys{/mmz/auto/replicate}{
1713   expanded/.code={\let\mmz@auto@replicate@expansion\@firstofone},
1714 }

```

to context This outer handler appends the original definition of the handled command to the `\mmz@auto@tocontext` text. The `\expandafter` are there to expand `\AdviceName` once before fully expanding `\AdviceGetOriginalCname`.

```

1715 \def\mmz@auto@tocontext{%
1716   \expanded{%
1717     \noexpand\pgfkeysvalueof{/mmz/context/.@cmd}%
1718     original "\AdviceNamespace" cname "\AdviceCname"={%
1719       \noexpand\expanded{%
1720         \noexpand\noexpand\noexpand\meaning
1721         \noexpand\AdviceCnameGetOriginal{\AdviceNamespace}{\AdviceCname}%
1722       }%
1723     }%
1724   }%
1725   \pgfeov
1726   \AdviceOriginal
1727 }

```

5.1 L^AT_EX-specific handlers

We handle cross-referencing (both the `\label` and the `\ref` side) and indexing. Note that the latter is a straightforward instance of replication.

```

1728 <(*latex)
1729 \mmzset{
1730   auto/.cd,
1731   ref/.style={outer handler=\mmz@auto@ref\mmzNoRef, run if memoizing},
1732   force ref/.style={outer handler=\mmz@auto@ref\mmzForceNoRef, run if memoizing},
1733 }
1734 \mmzset{
1735   auto=\ref{ref},
1736   auto=\pageref{ref},
1737   auto=\label{run if memoizing, outer handler=\mmz@auto@label},
1738   auto=\index{replicate, args=m, expanded},
1739 }

```

ref These keys install an outer handler which appends a cross-reference to the context. **force ref** does this even if the reference key is undefined, while **ref** aborts memoization in such a case — `\mmz@auto@ref` the idea is that it makes no sense to memoize when we expect the context to change in the next compilation anyway.

Any command taking a mandatory braced reference key argument potentially preceded by optional arguments of (almost) any kind may be submitted to these keys. This follows from the parameter list of `\mmz@auto@ref@i`, where `#2` grabs everything up to the first opening brace. The downside of the flexibility regarding the optional arguments is that unbraced single-token reference keys will cause an error, but as such usages of `\ref` and friends should be virtually inexistent, we let the bug stay.

`#1` should be either `\mmzNoRef` or `\mmzForceNoRef`. `#2` will receive any optional arguments of `\ref` (or `\pageref`, or whatever), and `#3` in `\mmz@auto@ref@i` is the cross-reference key.

```

1740 \def\mmz@auto@ref#1#2#{\mmz@auto@ref@i#1{#2}}
1741 \def\mmz@auto@ref@i#1#2#3{%
1742   #1{#3}%
1743   \AdviceOriginal#2{#3}%
1744 }

```

`\mmzForceNoRef` These macros do the real job in the outer handlers for cross-referencing, but it might be useful to have them publicly available. `\mmzForceNoRef` appends the reference key to the context. `\mmzNoRef` only does that if the reference is defined, otherwise it aborts the memoization.

```

1745 \def\mmzForceNoRef#1{%
1746   \mmz@mtoc@csname{r@#1}%
1747   \ignorespaces
1748 }
1749 \def\mmzNoRef#1{%
1750   \ifcsundef{r@#1}{\mmzAbort}{\mmzForceNoRef{#1}}%
1751   \ignorespaces
1752 }

```

`refrange` Let's rinse and repeat for reference ranges. The code is virtually the same as above, but we grab two reference key arguments (#3 and #4) in the final macro.

```

\mmz@auto@refrange
1753 \mmzset{
1754   auto/.cd,
1755   refrange/.style={outer handler=\mmz@auto@refrange\mmzNoRef,
1756     bailout handler=\relax, run if memoizing},
1757   force refrange/.style={outer handler=\mmz@auto@refrange\mmzForceNoRef,
1758     bailout handler=\relax, run if memoizing},
1759 }

1760 \def\mmz@auto@refrange#1#2#{\mmz@auto@refrange@i#1{#2}}
1761 \def\mmz@auto@refrange@i#1#2#3#4{%
1762   #1{#3}%
1763   #1{#4}%
1764   \AdviceOriginal#2{#3}{#4}%
1765 }

```

`multiref` And one final time, for “multi-references”, such as `cleveref`'s `\cref`, which can take a comma-separated list of reference keys in the sole argument. Again, only the final macro is any different, this time distributing #1 (`\mmzNoRef` or `\mmzForceNoRef`) over #3 by `\forcsvlist`.

```

1766 \mmzset{
1767   auto/.cd,
1768   multiref/.style={outer handler=\mmz@auto@multiref\mmzNoRef,
1769     bailout handler=\relax, run if memoizing},
1770   force multiref/.style={outer handler=\mmz@auto@multiref\mmzForceNoRef,
1771     bailout handler=\relax, run if memoizing},
1772 }
1773 \def\mmz@auto@multiref#1#2#{\mmz@auto@multiref@i#1{#2}}
1774 \def\mmz@auto@multiref@i#1#2#3{%
1775   \forcsvlist{#1}{#3}%
1776   \AdviceOriginal#2{#3}%
1777 }

```

`\mmz@auto@label` The outer handler for `\label` must be defined specifically for this command. The generic replicating handler is not enough here, as we need to replicate both the invocation of `\label` and the definition of `\@currentlabel`.

```

1778 \def\mmz@auto@label#1{%
1779   \xtoksapp\mmzCCMemo{%
1780     \noexpand\mmzLabel{#1}{\expandonce\@currentlabel}%
1781   }%
1782   \AdviceOriginal{#1}%
1783 }

```

`\mmzLabel` This is the macro that `\label`'s handler writes into the cc-memo. The first argument is the reference key; the second argument is the value of `\@currentlabel` at the time of invocation `\label` during memoization, which this macro temporarily restores.

```

1784 \def\mmzLabel#1#2{%
1785   \begingroup
1786   \def\@currentlabel{#2}%
1787   \label{#1}%
1788   \endgroup
1789 }
1790 </latex>

```

6 Support for various classes and packages

```

1791 <(*latex)>
1792 \AddToHook{shipout/before}[memoize]{\global\advance\mmzExtraPages-1\relax}
1793 \AddToHook{shipout/after}[memoize]{\global\advance\mmzExtraPages1\relax}
1794 \mmzset{auto=\DiscardShipoutBox{
1795   outer handler=\global\advance\mmzExtraPages1\relax\AdviceOriginal}}
1796 </latex>

```

Utility macro for clarity below. #1 is the name of the package which should be loaded (used with L^AT_EX) and #2 is the name of the command which should be defined (used with plain T_EX and ConT_EXt) for #3 to be executed at the beginning of the document. We make sure that we can use #1 etc. inside #3.

```

1797 \def\mmz@if@package@loaded#1#2#3{%
1798   \mmzset{%
1799     begindocument/before/.append code={%
1800 <latex>       \@ifpackageloaded{#1}{%
1801 <plain, context>   \ifdefined#2%
1802                   #3%
1803 <plain, context>   \fi
1804 <latex>       }{}%
1805   }%
1806 }%
1807 }

```

6.1 PGF

```

1808 \mmz@if@package@loaded{pgf}{%
1809 <plain>   \pgfpicture
1810 <context> \startpgfpicture
1811 }{%
1812   \def\mmzPgfAtBeginMemoization{%
1813     \edef\mmz@pgfpictureid{%
1814       \the\pgf@picture@serial@count
1815     }%
1816   }%
1817   \def\mmzPgfAtEndMemoization{%
1818     \edef\mmz@temp{%
1819       \the\numexpr\pgf@picture@serial@count-\mmz@pgfpictureid\relax
1820     }%
1821     \ifx\mmz@temp=0
1822     \else
1823       \xtoksapp\mmzCCMemo{%
1824         \unexpanded{%
1825           \global\expandafter\advance\csname pgf@picture@serial@count\endcsname
1826         }%
1827         \mmz@temp
1828       }%
1829     \fi
1830   }%
1831   \mmzset{%
1832     at begin memoization=\mmzPgfAtBeginMemoization,
1833     at end memoization=\mmzPgfAtEndMemoization,
1834 } }%

```

6.2 TikZ

In this section, we activate TikZ support (the collector is defined by Advice). All the action happens at the end of the preamble, so that we can detect whether TikZ was loaded (regardless of whether Memoize was loaded before TikZ, or vice versa), but still input the definitions.

```
1836 \mmz@if@package@loaded{tikz}{\tikz}{%
1837   \input advice-tikz.code.tex
```

We define and activate the automemoization handlers for the TikZ command and environment.

```
1838 \mmzset{%
1839   auto={tikzpicture}{memoize},
1840   auto=\tikz{memoize, collector=\AdviceCollectTikZArguments},
1841 }%
1842 }
```

6.3 Forest

Forest will soon feature extensive memoization support, but for now, let's just enable the basic, single extern externalization. Command `\Forest` is defined using `xparse`, so `args` is unnecessary.

```
1843 <*/latex>
1844 \mmz@if@package@loaded{forest}{\Forest}{%
1845   \mmzset{
1846     auto={forest}{memoize},
1847     auto=\Forest{memoize},
1848   }%
1849 }
1850 </latex>
```

6.4 Beamer

The Beamer code is explained in ^M§4.2.4.

```
1851 <*/latex>
1852 \AddToHook{begindocument/before}{\@ifclassloaded{beamer}{%
1853   \mmzset{per overlay/.style={
1854     /mmz/context={%
1855       overlay=\csname beamer@overlaynumber\endcsname,
1856       pauses=\ifmemoizing
1857         \mmzBeamerPauses
1858       \else
1859         \expandafter\the\csname c@beamerpauses\endcsname
1860       \fi
1861     },
1862     /mmz/at begin memoization={%
1863       \xdef\mmzBeamerPauses{\the\c@beamerpauses}%
1864       \xtoksapp\mmzCMemo{%
1865         \noexpand\mmzSetBeamerOverlays{\mmzBeamerPauses}{\beamer@overlaynumber}}%
1866       \gtoksapp\mmzCCMemo{%
1867         \only<\mmzBeamerOverlays>{}}%
1868     },
1869     /mmz/at end memoization={%
1870       \xtoksapp\mmzCCMemo{%
1871         \noexpand\setcounter{beamerpauses}{\the\c@beamerpauses}}%
1872     },
1873     /mmz/per overlay/.code={},
1874   }}
1875 \def\mmzSetBeamerOverlays#1#2{%
1876   \ifnum\c@beamerpauses=#1\relax
1877     \gdef\mmzBeamerOverlays{#2}%
1878     \ifnum\beamer@overlaynumber<#2\relax \mmz@temptrue \else \mmz@tempfalse \fi
```



```

1879 \else
1880 \mmz@temptrue
1881 \fi
1882 \ifmmz@temp
1883 \appto\mmzAtBeginMemoization{%
1884 \gtoksapp\mmzCMemo{\mmzSetBeamerOverlays{#1}{#2}}}%
1885 \fi
1886 }%
1887 }{}%
1888 </latex>

```

6.5 Biblatex

```

1889 <*latex>
1890 \mmzset{
1891 biblatex/.code={%
1892 \mmz@if@package@loaded{biblatex}{}{%
1893 \input memoize-biblatex.code.tex
1894 \mmzset{#1}%
1895 }%
1896 },
1897 }
1898 </latex>
1899 </mmz>
1900 <*biblatex>
1901 \edef\memoizeresetatcatcode{\catcode`\noexpand\@the\catcode`\@relax}%
1902 \catcode`\@=11
1903 \mmzset{%

```

Advise macro `\entry` occurring in .bb1 files to collect the entry, verbatim. `args`: `m` = citation key, `&&{...}u` = the entry, verbatim, braced — so `\blx@bbl@entry` will receive two mandatory arguments.

```

1904 auto=\blx@bbl@entry{
1905 inner handler=\mmz@biblatex@entry,
1906 args={%
1907 m%
1908 &&{\collargsVerb
1909 \collargsAppendExpandablePostprocessor{\the\collargsArg}}%
1910 }u{\endentry}%
1911 },

```

No braces around the collected arguments, as each is already braced on its own.

```

1912 raw collector options=\collargsReturnPlain,
1913 },

```

`cite` Define handlers for citation commands.

```

volcite
cites
volcites
1914 auto/cite/.style={
1915 run conditions=\mmz@biblatex@cite@rc,
1916 outer handler=\mmz@biblatex@cite@outer,
1917 args=l*m,
1918 raw collector options=\mmz@biblatex@def@star\collargsReturnNo,
1919 inner handler=\mmz@biblatex@cite@inner,
1920 },

```

We need a dedicated `volcite` even though `\volcite` executes `\cite` because otherwise, we would end up with `\cite{volume}{key}` in the cc-memo when `biblatex ccmemo cite=replicate`.

```

1921 auto/volcite/.style={
1922 run if memoizing,
1923 outer handler=\mmz@biblatex@cite@outer,
1924 args=lml*m,
1925 raw collector options=\mmz@biblatex@def@star\collargsReturnNo,

```



```

1926   inner handler=\mmz@biblatex@cite@inner,
1927 },
1928 auto/cites/.style={
1929   run conditions=\mmz@biblatex@cite@rc,
1930   outer handler=\mmz@biblatex@cites@outer,
1931   args=l*m,
1932   raw collector options=\mmz@biblatex@def@star\collargsClearArgsfalse\collargsReturnNo,
1933   inner handler=\mmz@biblatex@cites@inner,
1934 },
1935 auto/volcites/.style={
1936   run if memoizing,
1937   outer handler=\mmz@biblatex@cites@outer,
1938   args=lml*m,
1939   raw collector options=\mmz@biblatex@def@star\collargsClearArgsfalse\collargsReturnNo,
1940   inner handler=\mmz@biblatex@cites@inner,
1941 },

```

biblatex ccmemo cite What to put into the cc-memo, \nocite or the handled citation command?

```

1942 biblatex ccmemo cite/.is choice,
1943 biblatex ccmemo cite/nocite/.code={%
1944   \let\mmz@biblatex@do@ccmemo\mmz@biblatex@do@nocite
1945 },
1946 biblatex ccmemo cite/replicate/.code={%
1947   \let\mmz@biblatex@do@ccmemo\mmz@biblatex@do@replicate
1948 },
1949 }%

```

\mmz@biblatex@entry This macro stores the MD5 sum of the \entry when reading the .bb1 file.

```

1950 \def\mmz@biblatex@entry#1#2{%
1951   \edef\mmz@temp{\pdf@mdfivesum{#2}}%
1952   \global\cslet\mmz@bb1@#1\mmz@temp
1953   \mmz@scantokens{\AdviceOriginal{#1}#2}%
1954 }

```

\mmz@biblatex@cite@rc Run if memoizing but not within a \volcite command. Applied to \cite(s).

```

1955 \def\mmz@biblatex@cite@rc{%
1956   \ifmemoizing

```

We cannot use the official \ifvolcite, or even the blx@volcite toggle it depends on, because these are defined/set within the next-citation hook, which is yet to be executed. So we depend on the internal detail that \volcite and friends redefine \blx@citeargs to \blx@volciteargs.

```

1957   \ifx\blx@citeargs\blx@volciteargs
1958   \else
1959     \AdviceRuntrue
1960   \fi
1961 \fi
1962 }

```

\mmz@biblatex@cite@outer Initialize the macro receiving the citation key(s), and execute the collector.

```

1963 \def\mmz@biblatex@cite@outer{%
1964   \gdef\mmz@biblatex@keys{}%
1965   \AdviceCollector
1966 }

```

`\mmz@biblatex@mark@citation@key` We *append* to `\mmz@biblatex@keys` to automatically collect all citation keys of a `\cites` command; note that we use this system for `\cite` as well.

```

1967 \def\mmz@biblatex@def@star{%
1968   \collargsAlias{*}{&&\mmz@biblatex@mark@citation@key}}%
1969 }
1970 \def\mmz@biblatex@mark@citation@key{%
1971   \collargsAppendPreprocessor{\xappto\mmz@biblatex@keys{,\the\collargsArg}}%
1972 }

```

`\mmz@biblatex@cite@inner`

```

1973 \def\mmz@biblatex@cite@inner{%

```

This macro puts the `\cites` reference keys into the context, and adds `\nocite`, or the handled citation command, to the cc-memo.

```

1974   \mmz@biblatex@do@context
1975   \mmz@biblatex@do@ccmemo
1976   \expandafter\AdviceOriginal\the\collargsArgs
1977 }
1978 \def\mmz@biblatex@do@context{%
1979   \expandafter\forcsvlist
1980     \expandafter\mmz@biblatex@do@context@one
1981     \expandafter{\mmz@biblatex@keys}%
1982 }
1983 \def\mmz@biblatex@do@context@one#1{%
1984   \mmz@mtoc@csname{\mmz@bbl@#1}%
1985   \ifcsdef{\mmz@bbl@#1}{\mmz@Abort}%
1986 }
1987 \def\mmz@biblatex@do@nocite{%
1988   \xtoksapp\mmz@CCMemo{%
1989     \noexpand\nocite{\mmz@biblatex@keys}%
1990   }%
1991 }
1992 \def\mmz@biblatex@do@replicate{%
1993   \xtoksapp\mmz@CCMemo{%
1994     {%
1995       \nullfont

```

It is ok to use `\AdviceName` here, as the cc-memo is never input during memoization.

```

1996     \expandonce\AdviceName\the\collargsArgs
1997   }%
1998 }%
1999 }
2000 \let\mmz@biblatex@do@ccmemo\mmz@biblatex@do@nocite

```

`\mmz@biblatex@cites@outer` Same as for `\cite`, but we iterate the collector as long as the arguments continue.

```

2001 \def\mmz@biblatex@cites@outer{%
2002   \global\collargsArgs{}%
2003   \gdef\mmz@biblatex@keys{}%
2004   \AdviceCollector
2005 }
2006 \def\mmz@biblatex@cites@inner{%
2007   \futurelet\mmz@temp\mmz@biblatex@cites@inner@again
2008 }

```

If the following token is an opening brace or bracket, the multicite arguments continue.

```

2009 \def\mmz@biblatex@cites@inner@again{%
2010   \mmz@tempfalse
2011   \ifx\mmz@temp\bgroup

```

```

2012 \mmz@temptrue
2013 \else
2014 \ifx\mmz@temp[%]
2015 \mmz@temptrue
2016 \fi
2017 \fi
2018 \ifmmz@temp
2019 \expandafter\AdviceCollector
2020 \else
2021 \expandafter\mmz@biblatex@cites@inner@finish
2022 \fi
2023 }
2024 \def\mmz@biblatex@cites@inner@finish{%
2025 \mmz@biblatex@do@context
2026 \mmz@biblatex@do@ccmemo
2027 \expandafter\AdviceOriginal\the\collargsArgs
2028 }

```

[Advise](#) the citation commands.

```

2029 \mmzset{
2030 auto=\cite{cite},
2031 auto=\Cite{cite},
2032 auto=\parencite{cite},
2033 auto=\Parencite{cite},
2034 auto=\footcite{cite},
2035 auto=\footcitetext{cite},
2036 auto=\textcite{cite},
2037 auto=\Textcite{cite},
2038 auto=\smartcite{cite},
2039 auto=\Smartcite{cite},
2040 auto=\supercite{cite},
2041 auto=\cites{cites},
2042 auto=\Cites{cites},
2043 auto=\parencites{cites},
2044 auto=\Parenccites{cites},
2045 auto=\footcites{cites},
2046 auto=\footcitetexts{cites},
2047 auto=\smartcites{cites},
2048 auto=\Smartcites{cites},
2049 auto=\textcites{cites},
2050 auto=\Textcites{cites},
2051 auto=\supercites{cites},
2052 auto=\autocite{cite},
2053 auto=\Autocite{cite},
2054 auto=\autocites{cites},
2055 auto=\Autocites{cites},
2056 auto=\citeauthor{cite},
2057 auto=\Citeauthor{cite},
2058 auto=\citetitle{cite},
2059 auto=\citeyear{cite},
2060 auto=\citedate{cite},
2061 auto=\citeurl{cite},
2062 auto=\nocite{cite},
2063 auto=\fullcite{cite},
2064 auto=\footfullcite{cite},
2065 auto=\volcite{volcite},
2066 auto=\Volcite{volcite},
2067 auto=\volcites{volcites},
2068 auto=\Volcites{volcites},
2069 auto=\pvolcite{volcite},
2070 auto=\Pvolcite{volcite},
2071 auto=\pvolcites{volcites},

```

```

2072 auto=\Pvolcites{volcites},
2073 auto=\fvolcite{volcite},
2074 auto=\Fvolcite{volcite},
2075 auto=\fvolcites{volcites},
2076 auto=\Fvolcites{volcites},
2077 auto=\ftvolcite{volcite},
2078 auto=\ftvolcites{volcites},
2079 auto=\Ftvolcite{volcite},
2080 auto=\Ftvolcites{volcites},
2081 auto=\svolcite{volcite},
2082 auto=\Svolcite{volcite},
2083 auto=\svolcites{volcites},
2084 auto=\Svolcites{volcites},
2085 auto=\tvolcite{volcite},
2086 auto=\Tvolcite{volcite},
2087 auto=\tvolcites{volcites},
2088 auto=\Tvolcites{volcites},
2089 auto=\avolcite{volcite},
2090 auto=\Avolcite{volcite},
2091 auto=\avolcites{volcites},
2092 auto=\Avolcites{volcites},
2093 auto=\notecite{cite},
2094 auto=\Notecite{cite},
2095 auto=\pnotecite{cite},
2096 auto=\Pnotecite{cite},
2097 auto=\fnotecite{cite},

```

Similar to `volcite`, these commands must be handled specifically in order to function correctly with `biblatex ccmemo cite=replicate`.

```

2098 auto=\citenamelist{cite, args=1*mlm},
2099 auto=\citelist{cite, args=1*mlm},
2100 auto=\citefield{cite, args=1*mlm},
2101 }
2102 \memoizeresetcatcode
2103 </biblatex>
2104 <*mmz>

```

7 Initialization

`begindocument/before` These styles contain the initialization and the finalization code. They were populated throughout the source. Hook `begindocument/before` contains the package support code, which must be loaded while still in the preamble. Hook `begindocument` contains the initialization code whose execution doesn't require any particular timing, as long as it happens at the beginning of the document. Hook `begindocument/end` is where the commands are activated; this must crucially happen as late as possible, so that we successfully override foreign commands (like `hyperref`'s definitions). In \LaTeX , we can automatically execute these hooks at appropriate places:

```

2105 <*latex>
2106 \AddToHook{begindocument/before}{\mmzset{begindocument/before}}
2107 \AddToHook{begindocument}{\mmzset{begindocument}}
2108 \AddToHook{begindocument/end}{\mmzset{begindocument/end}}
2109 \AddToHook{enddocument/afterlastpage}{\mmzset{enddocument/afterlastpage}}
2110 </latex>

```

In plain \TeX , the user must execute these hooks manually; but at least we can group them together and given them nice names. Provisionally, manual execution is required in ConTeXt as well, as I'm not sure where to execute them — please help!

```

2111 <*plain, context>

```

```

2112 \mmzset{
2113   begin document/.style={begindocument/before, begindocument, begindocument/end},
2114   end document/.style={enddocument/afterlastpage},
2115 }
2116 </plain, context>

```

We clear the hooks after executing the last of them.

```

2117 \mmzset{
2118   begindocument/end/.append style={
2119     begindocument/before/.code={},
2120     begindocument/.code={},
2121     begindocument/end/.code={},
2122   }
2123 }

```

Formats other than plain T_EX need a way to prevent extraction during package-loading.

```

2124 <!plain> \mmzset{extract/no/.code={}}

```

memoize.cfg Load the configuration file. Note that **nomemoize** must input this file as well, because any special memoization-related macros defined by the user should be available; for example, my **memoize.cfg** defines `\ifregion` (see ^M§2.6).

```

2125 </mmz>
2126 <mmz, nommz> \InputIfFileExists{memoize.cfg}{}{}
2127 <*mmz>

```

For formats other than plain T_EX, we also save the current (initial or **memoize.cfg**-set) value of **extract**, so that we can restore it when package options include **extract=no**. Then, **extract** can be called without an argument in the preamble, triggering extraction using this method; this is useful e.g. if Memoize is compiled into a format.

```

2128 <!plain> \let\mmz@initial@extraction@method\mmz@extraction@method

```

Process the package options (except in plain T_EX).

```

2129 <*latex>
2130 \DeclareUnknownKeyHandler[mmz]{%
2131   \expanded{\noexpand\pgfqkeys{/mmz}{#1\IfBlankF{#2}{={#2}}}}
2132 \ProcessKeyOptions[mmz]
2133 </latex>
2134 <context> \expandafter\mmzset\expandafter{\currentmoduleparameters}

```

In L^AT_EX, **nomemoize** has to process package options as well, otherwise L^AT_EX will complain.

```

2135 </mmz>
2136 <*latex & nommz>
2137 \DeclareUnknownKeyHandler[mmz]{}
2138 \ProcessKeyOptions[mmz]
2139 </latex & nommz>

```

Extern extraction We redefine **extract** to immediately trigger extraction. This is crucial in plain T_EX, where extraction must be invoked after loading the package, but also potentially useful in other formats when package options include **extract=no**.

```

2140 <*mmz>
2141 \mmzset{
2142   extract/.is choice,
2143   extract/.default=\mmz@extraction@method,

```

But only once:

```

2144   extract/.append style={

```

```

2145     extract/.code={\PackageError{memoize}{Key "extract" was invoked twice.}{In
2146         principle, externs should be extracted only once.  If you really want
2147         to extract again, execute "extract/<method>".}},
2148 },

```

In formats other than plain T_EX, we remember the current `extract` code and then trigger the extraction.

```

2149 <!\plain> /utils/exec={\pgfkeysgetvalue{/mmz/extract/.@cmd}\mmz@temp@extract},
2150 <!\plain> extract=\mmz@extraction@method,
2151 }

```

Option `extract=no` (which only exists in formats other than plain T_EX) should allow for an explicit invocation of `extract` in the preamble.

```

2152 <*\plain>
2153 \def\mmz@temp{no}
2154 \ifx\mmz@extraction@method\mmz@temp
2155     \pgfkeyslet{/mmz/extract/.@cmd}\mmz@temp@extract
2156     \let\mmz@extraction@method\mmz@initial@extraction@method
2157 \fi
2158 \let\mmz@temp@extract\relax
2159 </!\plain>

```

Memoize was not really born for the draft mode, as it cannot produce new externs there. But we don't want to disable the package, as utilization and pure memoization are still perfectly valid in this mode, so let's just warn the user.

```

2160 \ifnum\pdf@draftmode=1
2161     \PackageWarning{memoize}{No externalization will be performed in the draft mode}%
2162 \fi
2163 </mmz>

```

Several further things which need to be defined as dummies in `nomemoize/memoizable`.

```

2164 <*\nommz, mmzable & generic>
2165 \pgfkeys{%
2166     /handlers/.meaning to context/.code={},
2167     /handlers/.value to context/.code={},
2168 }
2169 \let\mmzAbort\relax
2170 \let\mmzUnmemoizable\relax
2171 \newcommand\IfMemoizing[2][\@secondoftwo]{
2172 \let\mmzNoRef\@gobble
2173 \let\mmzForceNoRef\@gobble
2174 \newtoks\mmzContext
2175 \newtoks\mmzContextExtra
2176 \newtoks\mmzCMemo
2177 \newtoks\mmzCCMemo
2178 \newcount\mmzExternPages
2179 \newcount\mmzExtraPages
2180 \let\mmzTracingOn\relax
2181 \let\mmzTracingOff\relax
2182 </nommz, mmzable & generic>

```

The end of `memoize`, `nomemoize` and `memoizable`.

```

2183 <*\mmz, nommz, mmzable>
2184 <plain> \resetatcatcode
2185 <context> \stopmodule
2186 <context> \protect
2187 </mmz, nommz, mmzable>

```

That's all, folks!

8 Auxiliary packages

8.1 Extending commands and environments with Advice

```
2188 *main
2189 <latex> \ProvidesPackage{advice}[2024/01/02 v1.1.0 Extend commands and environments]
2190 <context> %D \module[
2191 <context> %D     file=t-advice.tex,
2192 <context> %D     version=1.1.0,
2193 <context> %D     title=Advice,
2194 <context> %D     subtitle=Extend commands and environments,
2195 <context> %D     author=Saso Zivanovic,
2196 <context> %D     date=2024-01-02,
2197 <context> %D     copyright=Saso Zivanovic,
2198 <context> %D     license=LPPL,
2199 <context> %D ]
2200 <context> \writestatus{loading}{ConTeXt User Module / advice}
2201 <context> \unprotect
2202 <context> \startmodule[advice]
```

Required packages

```
2203 <plain, context> \input miniltx
2204 <latex> \RequirePackage{collargs}
2205 <plain> \input collargs
2206 <context> \input t-collargs
```

In L^AT_EX, we also require `xparse`. Even though `\NewDocumentCommand` and friends are integrated into the L^AT_EX kernel, `\GetDocumentCommandArgSpec` is only available through `xparse`.

```
2207 <latex> \RequirePackage{xparse}
```

8.1.1 Installation into a keypath

.install advice This handler installs the advising mechanism into the handled path, which we shall henceforth also call the (advice) namespace.

```
2208 \pgfkeys{
2209   /handlers/.install advice/.code={%
2210     \edef\auto@install@namespace{\pgfkeyscurrentpath}%
2211     \def\advice@install@setupkey{advice}%
2212     \def\advice@install@activation{immediate}%
2213     \pgfqkeys{/advice/install}{#1}%
2214     \expanded{\noexpand\advice@install
2215       {\auto@install@namespace}%
2216       {\advice@install@setupkey}%
2217       {\advice@install@activation}%
2218     }%
2219   },
```

setup key These keys can be used in the argument of `.install advice` to configure the installation. By **activation** default, the setup key is `advice` and activation is `immediate`.

```
2220   /advice/install/.cd,
2221   setup key/.store in=\advice@install@setupkey,
2222   activation/.is choice,
2223   activation/.append code=\def\advice@install@activation{#1},
2224   activation/immediate/.code={},
2225   activation/deferred/.code={},
2226 }
```

#1 is the installation keypath (in Memoize, `/mmz`); #2 is the setup key name (in Memoize, `auto`, and this is why we document it as such); #3 is the initial activation regime.

```
2227 \def\advice@install#1#2#3{%
```

Switch to the installation keypath.

```
2228 \pgfqkeys{#1}{%
```

auto These keys submit a command or environment to advising. The namespace is hard-coded into
auto csname these keys via #1; their arguments are the command/environment (cs)name, and setup keys
auto key belonging to path *<installation keypath>/\meta{setup key name}*.

```
auto'
auto csname' 2229 #2/.code 2 args={%
```

auto key' Call the internal setup macro, wrapping the received keylist into a **pgfkeys** invocation.

```
2230 \AdviceSetup{#1}{#2}{##1}{\pgfqkeys{#1/#2}{##2}}%
```

Activate if not already activated (this can happen when updating the configuration). Note we don't call **\advice@activate** directly, but use the public keys; in this way, activation is automatically deferred if so requested. (We don't use **\pgfkeysalso** to allow **auto** being called from any path.)

```
2231 \pgfqkeys{#1}{try activate, activate={##1}}%
2232 },
```

A variant without activation.

```
2233 #2'/.code 2 args={%
2234 \AdviceSetup{#1}{#2}{##1}{\pgfqkeys{#1/#2}{##2}}%
2235 },
2236 #2 csname/.style 2 args={
2237 #2/.expand once=\expandafter{\csname ##1\endcsname}{##2},
2238 },
2239 #2 csname'/.style 2 args={
2240 #2'/.expand once=\expandafter{\csname ##1\endcsname}{##2},
2241 },
2242 #2 key/.style 2 args={
2243 #2/.expand once=%
2244 \expandafter{\csname pgfk@##1/.@cmd\endcsname}%
2245 {collector=\advice@pgfkeys@collector,##2},
2246 },
2247 #2 key'/.style 2 args={
2248 #2'/.expand once=%
2249 \expandafter{\csname pgfk@##1/.@cmd\endcsname}%
2250 {collector=\advice@pgfkeys@collector,##2},
2251 },
```

activation This key, residing in the installation keypath, forwards the request to the **/advice** path **activation** subkeys, which define **activate** and **friends** in the installation keypath. Initially, the activation regime is whatever the user has requested using the **.install advice** argument (here #3).

```
2252 activation/.style={/advice/activation/##1={#1}},
2253 activation=#3,
```

activate deferred The deferred activations are collected in this style, see section **refsec:code:advice:activation** for details.

```
2254 activate deferred/.code={},
```

activate csname For simplicity of implementation, the **csname** versions of **activate** and **deactivate** accept a
deactivate csname single *<csname>*. This way, they can be defined right away, as they don't change with the type of activation (immediate vs. deferred).

```
2255 activate csname/.style={activate/.expand once={\csname##1\endcsname}},
2256 deactivate csname/.style={deactivate/.expand once={\csname##1\endcsname}},
```


activate key (De)activation of pgfkeys keys. Accepts a list of key names, requires full key names.

```

deactivate key
2257   activate key/.style={activate@key={#1/activate}{##1}},
2258   deactivate key/.style={activate@key={#1/deactivate}{##1}},
2259   activate@key/.code n args=2{%
2260     \def\advice@temp{%
2261       \def\advice@do####1{%
2262         \eappto\advice@temp{,\expandonce{\csname pgfk@####1/.@cmd\endcsname}}}%
2263     \forcsvlist\advice@do{##2}%
2264     \pgfkeysalso{##1/.expand once=\advice@temp}%
2265   },

```

The rest of the keys defined below reside in the auto subfolder of the installation keypath.

```

2266   #2/.cd,

```

run conditions These keys are used to setup the handling of the command or environment. The **outer handler** storage macros (`\AdviceRunConditions` etc.) have public names as they also play a crucial role in the handler definitions, see section 8.1.3.

```

collector
  args
2267   run conditions/.store in=\AdviceRunConditions,
2268   bailout handler/.store in=\AdviceBailoutHandler,
collector options
2269   outer handler/.store in=\AdviceOuterHandler,
clear collector options
2270   collector/.store in=\AdviceCollector,
raw collector options
2271   collector options/.code={\appto\AdviceCollectorOptions{,##1}},
clear raw collector options
2272   clear collector options/.code={\def\AdviceCollectorOptions{}},
inner handler
2273   raw collector options/.code={\appto\AdviceRawCollectorOptions{##1}},
options
2274   clear raw collector options/.code={\def\AdviceRawCollectorOptions{}},
clear options
2275   args/.store in=\AdviceArgs,
2276   inner handler/.store in=\AdviceInnerHandler,
2277   options/.code={\appto\AdviceOptions{,##1}},
2278   clear options/.code={\def\AdviceOptions{}},

```

A user-friendly way to set options: any unknown key is an option.

```

2279   .unknown/.code={%
2280     \eappto{\AdviceOptions}{,\pgfkeyscurrentname={\unexpanded{##1}}}%
2281   },

```

The default values of the keys, which equal the initial values for commands, as assigned by `\advice@setup@init@command`.

```

2282   run conditions/.default=\AdviceRuntrue,
2283   bailout handler/.default=\relax,
2284   outer handler/.default=\AdviceCollector,
2285   collector/.default=\advice@CollectArgumentsRaw,
2286   collector options/.value required,
2287   raw collector options/.value required,
2288   args/.default=\advice@noargs,
2289   inner handler/.default=\advice@error@noinnerhandler,
2290   options/.value required,

```

reset This key resets the advice settings to their initial values, which depend on whether we're handling a command or environment.

```

2291   reset/.code={\csname\advice@setup@init@AdviceType\endcsname},

```

after setup The code given here will be executed once we exit the setup group. `integrated driver` of Memoize uses it to declare a conditional.

```

2292   after setup/.code={\appto\AdviceAfterSetup{##1}},

```

In L^AT_EX, we finish the installation by submitting `\begin`; the submission is funky, because the run conditions handler actually hacks the standard handling procedure. Note that if `\begin` is not activated, environments will not be handled, and that the automatic activation might be deferred.

```
2293 <latex>    #1/#2=\begin{run conditions=\advice@begin@rc},
2294    }%
2295 }
```

8.1.2 Submitting a command or environment

\AdviceSetup Macro `\advice@setup` is called by key `auto` to submit a command or environment to advising.
\AdviceName It receives four arguments: `#1` is the installation keypath / storage namespace; `#2` is the name of the setup key; `#3` is the submitted command or environment; `#4` is the setup code (which is only grabbed by `\advice@setup@i`).

Executing this macro defines macros `\AdviceName`, holding the control sequence of the submitted command or the environment name, and `\AdviceType`, holding `command` or `environment`; they are used to set up some initial values, and may be used by user-defined keys in the `auto` path, as well (see `/mmz/auto/noop` for an example). The macro then performs internal initialization, and finally calls the second part, `\advice@setup@i`, with the command's *storage* name as the first argument.

This macro also serves as the programmer's interface to `auto`, the idea being that an advanced user may write code `#4` which defined the settings macros (`\AdviceOuterHandler` etc.) without deploying `pgfkeys`. (Also note that activation at the end only occurs through the `auto` interface.)

```
2296 \def\AdviceSetup#1#2#3{%
```

Open a group, so that we allow for embedded `auto` invocations.

```
2297   \begin{group}
2298   \def\AdviceName{#3}%
2299   \advice@def@AdviceCname
```

Command, complain, or environment?

```
2300   \collargs@cs@cases{#3}{%
2301     \def\AdviceType{command}%
2302     \advice@setup@init@command
2303     \advice@setup@i{#3}{#1}{#3}%
2304   }{%
2305     \advice@error@advice@notcs{#1/#2}{#3}%
2306   }{%
2307     \def\AdviceType{environment}%
2308     \advice@setup@init@environment
2309 <latex>    \advice@setup@i{#3}%
2310 <plain>    \expandafter\advice@setup@i\expandafter{\csname #3\endcsname}%
2311 <context>    \expandafter\advice@setup@i\expandafter{\csname start#3\endcsname}%
2312             {#1}{#3}%
2313   }%
2314 }
```

The arguments of `\advice@setup@i` are a bit different than for `\advice@setup`, because we have inserted the storage name as `#1` above, and we lost the setup key name `#2`. Here, `#2` is the installation keypath / storage namespace, `#3` is the submitted command or environment; and `#4` is the setup code.

What is the difference between the storage name (`#1`) and the command/environment name (`#3`, and also the contents of `\AdviceName`), and why do we need both? For commands, there is actually no difference; for example, when submitting command `\foo`, we end up with `#1=#3=\foo`. And there is also no difference for L^AT_EX environments; when submitting environment `foo`, we get `#1=#3=foo`. But in plain T_EX, `#1=\foo` and `#3=foo`, and in ConT_EXt, `#1=\startfoo` and `#3=foo` — which should explain the guards and `\expandafters` above.

And why both #1 and #3? When a handled command is executed, it loads its configuration from a macro determined by the storage namespace and the (`\stringified`) storage name, e.g. `/mmz` and `\foo`. In plain `TEX` and `ConTEXt`, each environment is started by a dedicated command, `\foo` or `\startfoo`, so these control sequences (`\stringified`) must act as storage names. (Not so in `LATEX`, where an environment configuration is loaded by `\begin`'s handler, which can easily work with storage name `foo`. Even more, having `\foo` as an environment storage name would conflict with the storage name for the (environment-internal) command `\foo` — yes, we can submit either `foo` or `\foo`, or both, to advising.)

```
2315 \def\advice@setup@i#1#2#3#4{%
```

Load the current configuration of the handled command or environment — if it exists.

```
2316 \advice@setup@init@i{#2}{#1}%
2317 \advice@setup@init@I{#2}{#1}%
2318 \def\AdviceAfterSetup{}%
```

Apply the setup code/keys.

```
2319 #4%
```

Save the resulting configuration. This closes the group, because the config is saved outside it.

```
2320 \advice@setup@save{#2}{#1}%
2321 }
```

[Initialize](#) the configuration of a command or environment. Note that the default values of the keys equal the initial values for commands. Nothing would go wrong if these were not the same, but it's nice that the end-user can easily revert to the initial values.

```
2322 \def\advice@setup@init@common{%
2323 \def\AdviceRunConditions{\AdviceRuntrue}%
2324 \def\AdviceBailoutHandler{\relax}%
2325 \def\AdviceOuterHandler{\AdviceCollector}%
2326 \def\AdviceCollector{\advice@CollectArgumentsRaw}%
2327 \def\AdviceCollectorOptions{}%
2328 \def\AdviceInnerHandler{\advice@error@noinnerhandler}%
2329 \def\AdviceOptions{}%
2330 }
2331 \def\advice@setup@init@command{%
2332 \advice@setup@init@common
2333 \def\AdviceRawCollectorOptions{}%
2334 \def\AdviceArgs{\advice@noargs}%
2335 }
2336 \def\advice@setup@init@environment{%
2337 \advice@setup@init@common
2338 \edef\AdviceRawCollectorOptions{%
2339 \noexpand\collargsEnvironment{\AdviceName}%
```

When grabbing an environment body, the end-tag will be included. This makes it possible to have the same inner handler for commands and environments.

```
2340 \noexpand\collargsEndTagtrue
2341 }%
2342 \def\AdviceArgs{+b}%
2343 }
```

We need to initialize `\AdviceOuterHandler` etc. so that `\advice@setup@store` will work.

```
2344 \advice@setup@init@command
```

The configuration storage The remaining macros in this subsection deal with the configuration storage space, which is set up in a way to facilitate fast loading during the execution of handled commands and environments.

The configuration of a command or environment is stored in two parts: the first stage settings comprise the run conditions, the bailout handler and the outer handler; the second stage settings contain the rest. When a handled command is invoked, only the first stage settings are immediately loaded, for speed; the second stage settings are only loaded if the run conditions are satisfied.

\advice@init@i The two-stage settings are stored in control sequences `\advice@i<namespace>///<storage name>` and `\advice@I<namespace>///<storage name>`, respectively, and accessed using macros `\advice@init@i` and `\advice@init@I`.

Each setting storage macro contains a sequence of items, where each item is either of form `\def\AdviceSetting{<value>}`. This allows us store multiple settings in a single macro (rather than define each control-sequence-valued setting separately, which would use more string memory), and also has the consequence that we don't require the handlers to be defined when submitting a command (whether that's good or bad could be debated: as things stand, any typos in handler declarations will only yield an error once the handled command is executed).

```
2345 \def\advice@init@i#1#2{\csname advice@i#1//\string#2\endcsname}
2346 \def\advice@init@I#1#2{\csname advice@I#1//\string#2\endcsname}
```

We make a copy of these for setup; the originals might be swapped for tracing purposes.

```
2347 \let\advice@setup@init@i\advice@init@i
2348 \let\advice@setup@init@I\advice@init@I
```

\advice@setup@save To save the configuration at the end of the setup, we construct the storage macros out of `\AdviceRunConditions` and friends. Stage-one contains only `\AdviceRunConditions` and `\AdviceBailoutHandler`, so that `\advice@handle` can bail out as quickly as possible if the run conditions are not met.

```
2349 \def\advice@setup@save#1#2{%
2350   \expanded%
```

Close the group before saving. Note that `\expanded` has already expanded the settings macros.

```
2351   \endgroup
2352   \noexpand\csdef{advice@i#1//\string#2}{%
2353     \def\noexpand\AdviceRunConditions{\expandonce\AdviceRunConditions}%
2354     \def\noexpand\AdviceBailoutHandler{\expandonce\AdviceBailoutHandler}%
2355   }%
2356   \noexpand\csdef{advice@I#1//\string#2}{%
2357     \def\noexpand\AdviceOuterHandler{\expandonce\AdviceOuterHandler}%
2358     \def\noexpand\AdviceCollector{\expandonce\AdviceCollector}%
2359     \def\noexpand\AdviceRawCollectorOptions{%
2360       \expandonce\AdviceRawCollectorOptions}%
2361     \def\noexpand\AdviceCollectorOptions{\expandonce\AdviceCollectorOptions}%
2362     \def\noexpand\AdviceArgs{\expandonce\AdviceArgs}%
2363     \def\noexpand\AdviceInnerHandler{\expandonce\AdviceInnerHandler}%
2364     \def\noexpand\AdviceOptions{\expandonce\AdviceOptions}%
2365   }%
2366   \expandonce{\AdviceAfterSetup}%
2367 }%
2368 }
```

activation/immediate These two subkeys of `/advice/activation` install the immediate and the deferred activation code into the installation keypath. They are invoked by key `<installation keypath>/activation=<type>`.

Under the deferred activation regime, the commands are not (de)activated right away. Rather, the (de)activation calls are collected in style `activate deferred`, which should be

executed by the installation keypath owner, if and when they so desire. (Be sure to switch to `activation=immediate` before executing `activate deferred`, otherwise the activation will only be deferred once again.)

```

2369 \pgfkeys{
2370   /advice/activation/deferred/.style={
2371     #1/activate/.style={%
2372       activate deferred/.append style={#1/activate={##1}}},
2373     #1/deactivate/.style={%
2374       activate deferred/.append style={#1/deactivate={##1}}},
2375     #1/force activate/.style={%
2376       activate deferred/.append style={#1/force activate={##1}}},
2377     #1/try activate/.style={%
2378       activate deferred/.append style={#1/try activate={##1}}},
2379   },

```

activate The “real,” immediate `activate` and `deactivate` take a comma-separated list of commands or environments and (de)activate them. If `try activate` is in effect, no error is thrown upon failure.

deactivate

force activate If `force activate` is in effect, activation proceeds even if we already had the original definition;

try activate it does not apply to deactivation. These conditionals are set to false after every invocation of key (de)activate, so that they only apply to the immediately following (de)activate. (#1 below is the `<namespace>`; ##1 is the list of commands to be (de)activated.)

```

2380   /advice/activation/immediate/.style={
2381     #1/activate/.code={%
2382       \forcsvlist{\advice@activate{#1}}{##1}%
2383       \advice@activate@forcefalse
2384       \advice@activate@tryfalse
2385     },
2386     #1/deactivate/.code={%
2387       \forcsvlist{\advice@deactivate{#1}}{##1}%
2388       \advice@activate@forcefalse
2389       \advice@activate@tryfalse
2390     },
2391     #1/force activate/.is if=advice@activate@force,
2392     #1/try activate/.is if=advice@activate@try,
2393   },
2394 }
2395 \newif\ifadvice@activate@force
2396 \newif\ifadvice@activate@try

```

\advice@original@csname Activation replaces the original meaning of the handled command with our definition. We

\advice@original@cs store the original definition into control sequence `\advice@o<namespace>//<storage name>`

\AdviceGetOriginal (with a `\stringified <storage name>`). Internally, during (de)activation and handling, we access it using `\advice@original@csname` and `\advice@original@cs`. Publicly it should always be accessed by `\AdviceGetOriginal`, which returns the argument control sequence if that control sequence is not handled.

Using the internal command outside the handling context, we could fall victim to scenario such as the following. When we memoize something containing a `\label`, the produced cc-memo contains code eventually executing the original `\label`. If we called the original `\label` via the internal macro there, and the user `deactivated \label` on a subsequent compilation, the cc-memo would not call `\label` anymore, but `\relax`, resulting in a silent error. Using `\AdviceGetOriginal`, the original `\label` will be executed even when not activated.

However, not all is bright with `\AdviceGetOriginal`. Given an activated control sequence (#2), a typo in the namespace argument (#1) will lead to an infinite loop upon the execution of `\AdviceGetOriginal`. In the manual, we recommend defining a namespace-specific macro to avoid such typos.

```

2397 \def\advice@original@csname#1#2{advice@o#1//\string#2}
2398 \def\advice@original@cs#1#2{csname advice@o#1//\string#2\endcsname}

```

```

2399 \def\AdviceGetOriginal#1#2{%
2400   \ifcsname advice@o#1//\string#2\endcsname
2401     \expandonce{\csname advice@o#1//\string#2\expandafter\endcsname\expandafter}%
2402   \else
2403     \unexpanded\expandafter{\expandafter#2\expandafter}%
2404   \fi
2405 }

```

\AdviceCnameGetOriginal A version of `\AdviceGetOriginal` which accepts a control sequence name as the second argument.

```

2406 \begingroup
2407 \catcode`\/=0
2408 \catcode`\=12
2409 /gdef/advice@backslash@other{\}%
2410 /endgroup
2411 \def\AdviceCnameGetOriginal#1#2{%
2412   \ifcsname advice@o#1//\advice@backslash@other#2\endcsname
2413     \expandonce{\csname advice@o#1//\advice@backslash@other#2\expandafter\endcsname
2414       \expandafter}%
2415   \else
2416     \expandonce{\csname#2\expandafter\endcsname\expandafter}%
2417   \fi
2418 }

```

\advice@activate These macros execute either the command, or the environment (de)activator.

```

\advice@deactivate
2419 \def\advice@activate#1#2{%
2420   \collargs@cs@cases{#2}%
2421   {\advice@activate@cmd{#1}{#2}}%
2422   {\advice@error@activate@notcsorenv{#1}}%
2423   {\advice@activate@env{#1}{#2}}%
2424 }
2425 \def\advice@deactivate#1#2{%
2426   \collargs@cs@cases{#2}%
2427   {\advice@deactivate@cmd{#1}{#2}}%
2428   {\advice@error@activate@notcsorenv{de}{#1}}%
2429   {\advice@deactivate@env{#1}{#2}}%
2430 }

```

\advice@activate@cmd We are very careful when we're activating a command, because activating means rewriting its original definition. Configuration by `auto` did not touch the original command; activation will. So, the leitmotif of this macro: safety first. (`#1` is the namespace, and `#2` is the command to be activated.)

```

2431 \def\advice@activate@cmd#1#2{%

```

Is the command defined?

```

2432   \ifdef{#2}{%

```

Yes, the command is defined. Let's see if it's safe to activate it. We'll do this by checking whether we have its original definition in our storage. If we do, this means that we have already activated the command. Activating it twice would lead to the loss of the original definition (because the second activation would store our own redefinition as the original definition) and consequently an infinite loop (because once — well, if — the handler tries to invoke the original command, it will execute itself all over).

```

2433     \ifcsdef{\advice@original@csname{#1}{#2}}{%

```

Yes, we have the original definition, so the safety check failed, and we shouldn't activate again. Unless ... how does its current definition look like?

```

2434       \advice@if@our@definition{#1}{#2}{%

```


Well, the current definition of the command matches what we would put there ourselves. The command is definitely activated, and we refuse to activate again, as that would destroy the original definition.

```
2435 \advice@activate@error@activated{#1}{#2}{Command}{already}%
2436 }{%
```

We don't recognize the current definition as our own code (despite the fact that we have surely activated the command before, given the result of the first safety check). It appears that someone else was playing fast and loose with the same command, and redefined it after our activation. (In fact, if that someone else was another instance of Advice, from another namespace, forcing the activation will result in the loss of the original definition and the infinite loop.) So it *should* be safe to activate it (again) ... but we won't do it unless the user specifically requested this using `force activate`. Note that without `force activate`, we would be stuck in this branch, as we could neither activate (again) nor deactivate the command.

```
2437 \ifadvice@activate@force
2438 \advice@activate@cmd@do{#1}{#2}%
2439 \else
2440 \advice@activate@error@activated{#1}{#2}{Command}{already}%
2441 \fi
2442 }%
2443 }{%
```

No, we don't have the command's original definition, so it was not yet activated, and we may activate it.

```
2444 \advice@activate@cmd@do{#1}{#2}%
2445 }%
2446 }{%
2447 \advice@activate@error@undefined{#1}{#2}{Command}{}%
2448 }%
2449 }
```

`\advice@deactivate@cmd` The deactivation of a command follows the same template as activation, but with a different logic, and of course a different effect. In order to deactivate a command, both safety checks discussed above must be satisfied: we must have the command's original definition, *and* our redefinition must still reside in the command's control sequence — the latter condition prevents overwriting someone else's redefinition with the original command. As both conditions must be unavoidably fulfilled, `force activate` has no effect in deactivation (but `try activate` has).

```
2450 \def\advice@deactivate@cmd#1#2{%
2451 \ifdef{#2}{%
2452 \ifcsdef\advice@original@csname{#1}{#2}{%
2453 \advice@if@our@definition{#1}{#2}{%
2454 \advice@deactivate@cmd@do{#1}{#2}%
2455 }{%
2456 \advice@deactivate@error@changed{#1}{#2}%
2457 }%
2458 }{%
2459 \advice@activate@error@activated{#1}{#2}{Command}{not yet}%
2460 }%
2461 }{%
2462 \advice@activate@error@undefined{#1}{#2}{Command}{de}%
2463 }%
2464 }
```

`\advice@if@our@definition` This macro checks whether control sequence #2 was already activated (in namespace #1) in the sense that its current definition contains the code our activation would put there: `\advice@handle{#1}{#2}` (protected).


```

2465 \def\advice@if@our@definition#1#2{%
2466   \protected\def\advice@temp{\advice@handle{#1}{#2}}%
2467   \ifx#2\advice@temp
2468     \expandafter\@firstoftwo
2469   \else
2470     \expandafter\@secondoftwo
2471   \fi
2472 }

```

`\advice@activate@cmd@do` This macro saves the original command, and redefines its control sequence. Our redefinition must be `\protected` — even if the original command wasn’t fragile, our replacement certainly is. (Note that as we require ε -TeX anyway, we don’t have to pay attention to L^AT_EX’s robust commands by redefining their “inner” command. Protecting our replacement suffices.)

```

2473 \def\advice@activate@cmd@do#1#2{%
2474   \cslet{\advice@original@csname{#1}{#2}}#2%
2475   \protected\def#2{\advice@handle{#1}{#2}}%
2476   \PackageInfo{advice (#1)}{Activated command "\string#2"}%
2477 }

```

`\advice@deactivate@cmd@do` This macro restores the original command, and removes its definition from our storage — this also serves as a signal that the command is not activated anymore.

```

2478 \def\advice@deactivate@cmd@do#1#2{%
2479   \letcs#2{\advice@original@csname{#1}{#2}}%
2480   \csundef{\advice@original@csname{#1}{#2}}%
2481   \PackageInfo{advice (#1)}{Deactivated command "\string#2"}%
2482 }

```

8.1.3 Executing a handled command

`\advice@handle` An invocation of this macro is what replaces the original command and runs the whole shebang. The system is designed to bail out as quickly as necessary if the run conditions are not met (plus L^AT_EX’s `\begin` will receive a very special treatment for this reason).

We first check the run conditions, and bail out if they are not satisfied. Note that only the stage-one config is loaded at this point. It sets up the following macros (while they are public, neither the end user not the installation keypath owner should ever have to use them):

- `\AdviceRunConditions` executes `\AdviceRuntrue` if the command should be handled; set by `run conditions`.
- `\AdviceBailoutHandler` will be executed if the command will not be handled, after all; set by `bailout handler`.

```

2483 \def\advice@handle#1#2{%
2484   \advice@init@i{#1}{#2}%
2485   \AdviceRunfalse
2486   \AdviceRunConditions
2487   \advice@handle@rc{#1}{#2}%
2488 }

```

`\advice@handle@rc` We continue the handling in a new macro, because this is the point where the handler for `\begin` will hack into the regular flow of events.

```

2489 \def\advice@handle@rc#1#2{%
2490   \ifAdviceRun
2491     \expandafter\advice@handle@outer
2492   \else

```

Bailout is simple: we first execute the handler, and then the original command.

```

2493     \AdviceBailoutHandler
2494     \expandafter\advice@original@cs

```

```

2495 \fi
2496 {\#1}{\#2}%
2497 }

```

`\advice@handle@outer` To actually handle the command, we first setup some macros:

- `\AdviceNamespace` holds the installation keypath / storage name space.
- `\AdviceName` holds the control sequence of the handled command, or the environment name.
- `\AdviceReplaced` holds the “substituted” code. For commands, this is the same as `\AdviceName`. For environment `foo`, it equals `\begin{foo}` in L^AT_EX, `\foo` in plain T_EX and `\startfoo` in ConT_EXt.
- `\AdviceOriginal` executes the original definition of the handled command or environment.

```

2498 \def\advice@handle@outer#1#2{%
2499   \def\AdviceNamespace{#1}%
2500   \def\AdviceName{#2}%
2501   \advice@def@AdviceCname
2502   \let\AdviceReplaced\AdviceName
2503   \def\AdviceOriginal{\AdviceGetOriginal{#1}{#2}}%

```

We then load the stage-two settings. This defines the following macros:

- `\AdviceOuterHandler` will effectively replace the command, if it will be handled; set by outer handler.
- `\AdviceCollector` collects the arguments of the handled command, perhaps consulting `\AdviceArgs` to learn about its argument structure.
- `\AdviceRawCollectorOptions` contains the options which will be passed to the argument collector, in the “raw” format.
- `\AdviceCollectorOptions` contains the additional, user-specified options which will be passed to the argument collector.
- `\AdviceArgs` contains the `xparse`-style argument specification of the command, or equals `\advice@noargs` to signal that command was defined using `xparse` and that the argument specification should be retrieved automatically.
- `\AdviceInnerHandler` is called by the argument collector once it finishes its work. It receives all the collected arguments as a single (braced) argument.
- `\AdviceOptions` holds options which may be used by the outer or the inner handler; Advice does not need or touch them.

```

2504 \advice@init@I{#1}{#2}%

```

All prepared, we execute the outer handler.

```

2505 \AdviceOuterHandler
2506 }
2507 \def\advice@def@AdviceCname{%
2508   \begingroup
2509   \escapechar=-1
2510   \expandafter\expandafter\expandafter\endgroup
2511   \expandafter\expandafter\expandafter\def
2512   \expandafter\expandafter\expandafter\AdviceCname
2513   \expandafter\expandafter\expandafter{\expandafter\string\AdviceName}%
2514 }

```

`\ifAdviceRun` This conditional is set by the run conditions macro to signal whether we should run the outer (true) or the bailout (false) handler.

```

2515 \newif\ifAdviceRun

```

`\advice@CollectArgumentsRaw` This is the default collector, which will collect the argument using CollArgs’ command `\CollectArgumentsRaw`. It will provide that command with:

- the collector options, given in the raw format:
 - the caller (`\collargsCaller`),

- the raw options (`\AdviceRawCollectorOptions`), and
- the user options (`\AdviceRawCollectorOptions`, wrapped in `\collargsSet`;
- the argument specification `\AdviceArgs` of the handled command; and
- the inner handler `\AdviceInnerHandler` to execute after collecting the arguments; the inner handler receives the collected arguments as a single braced argument.

If the argument specification is not defined (either the user did not set it, or has reset it by writing `args` without a value), it is assumed that the handled command was defined by `xparse` and `\AdviceArgs` will be retrieved by `\GetDocumentCommandArgSpec`.

```

2516 \def\advice@CollectArgumentsRaw{%
2517   \AdviceIfArgs{}{%
2518     \expandafter\GetDocumentCommandArgSpec\expandafter{\AdviceName}%
2519     \let\AdviceArgs\ArgumentSpecification
2520   }%
2521   \expanded{%
2522     \noexpand\CollectArgumentsRaw{%
2523       \noexpand\collargsCaller{\expandonce\AdviceName}%
2524       \expandonce\AdviceRawCollectorOptions
2525       \ifdefempty\AdviceCollectorOptions{}{%
2526         \noexpand\collargsSet{\expandonce\AdviceCollectorOptions}%
2527       }%
2528     }%
2529     {\expandonce\AdviceArgs}%
2530     {\expandonce\AdviceInnerHandler}%
2531   }%
2532 }

```

`\AdviceIfArgs` If the value of `args` is “real”, i.e. an `xparse` argument specification, execute the first argument. If `args` was set to the special value `\advice@noargs`, signaling a command defined by `\NewDocumentCommand` or friends, execute the second argument. (Ok, in reality anything other than `\advice@noargs` counts as real “real”.)

```

2533 \def\advice@noargs@text{\advice@noargs}
2534 \def\AdviceIfArgs{%
2535   \ifx\AdviceArgs\advice@noargs@text
2536     \expandafter\@secondoftwo
2537   \else
2538     \expandafter\@firstoftwo
2539   \fi
2540 }

```

`\advice@pgfkeys@collector` A `pgfkeys` collector is very simple: the sole argument of the any key macro, regardless of the argument structure of the key, is everything up to `\pgfeov`.

```

2541 \def\advice@pgfkeys@collector#1\pgfeov{%
2542   \AdviceInnerHandler{#1}%
2543 }

```

8.1.4 Environments

`\advice@activate@env` Things are simple in `TEX` and `ConTEXt`, as their environments are really commands. So `\advice@deactivate@env` rather than activating environment name `#2`, we (de)activate command `\#2` or `\start#2`, depending on the format.

```

2544 <plain, context>
2545 \def\advice@activate@env#1#2{%
2546   \expanded{%
2547     \noexpand\advice@activate@cmd{#1}{\expandonce{\csname
2548 <context>      start%
2549       #2\endcsname}}}%
2550   }%
2551 }

```

```

2552 \def\advice@deactivate@env#1#2{%
2553   \expanded{%
2554     \noexpand\advice@deactivate@cmd{#1}{\expandonce{\csname
2555 <context>      start%
2556       #2\endcsname}}}%
2557   }%
2558 }
2559 </plain, context>

```

We activate commands by redefining them; that's the only way to do it. But we won't activate a L^AT_EX environment `foo` by redefining command `\foo`, where the user's definition for the start of the environment actually resides, as such a redefinition would be executed too late, deep within the group opened by `\begin`, following many internal operations and public hooks. We handle L^AT_EX environments by defining an outer handler for `\begin` (consequently, L^AT_EX environment support can be (de)activated by the user by saying `(de)activate=\begin`), and activating an environment will be nothing but setting a mark, by defining a dummy control sequence `\advice@original@csname{#1}{#2}`, which that handler will inspect. Note that `force activate` has no effect here.

```

2560 <(*latex)>
2561 \def\advice@activate@env#1#2{%
2562   \ifcsdef{\advice@original@csname{#1}{#2}}{%
2563     \advice@activate@error@activated{#1}{#2}{Environment}{already}%
2564   }{%
2565     \csdef{\advice@original@csname{#1}{#2}}{%
2566       \PackageInfo{advice (#1)}{Activated environment "#2"}%
2567     }%
2568   }
2569 \def\advice@deactivate@env#1#2{%
2570   \ifcsdef{\advice@original@csname{#1}{#2}}{%
2571     \csundef{\advice@original@csname{#1}{#2}}{%
2572   }{%
2573     \advice@activate@error@activated{#1}{#2}{Environment}{not yet}%
2574     \PackageInfo{advice (#1)}{Deactivated environment "#2"}%
2575   }%
2576 }

```

`\advice@begin@rc` This is the handler for `\begin`. It is very special, for speed. It is meant to be declared as the run conditions component, and it hacks into the normal flow of handling. It knows that after executing the run conditions macro, `\advice@handle` eventually (the tracing info may interrupt here as `#1`) continues by `\advice@handle@rc{\<namespace>}{\<handled control sequence>}`, so it grabs all these (`#2` is the `\<namespace>` and `#3` is the `\<handled control sequence>`, i.e. `\begin`) plus the environment name (`#4`).

```

2577 \def\advice@begin@rc#1\advice@handle@rc#2#3#4{%

```

We check whether environment `#4` is activated (in namespace `#2`) by inspecting whether activation dummy is defined. If it is not, we execute the original `\begin` (`\advice@original@cs{#2}{#3}`), followed by the environment name (`#4`). Note that we *don't* execute the environment's bailout handler here: we haven't checked its run conditions yet, as the environment is simply not activated.

```

2578   \ifcsname\advice@original@csname{#2}{#4}\endcsname
2579     \expandafter\advice@begin@env@rc
2580   \else
2581     \expandafter\advice@original@cs
2582   \fi
2583   {#2}{#3}{#4}%
2584 }

```

`\advice@begin@env@rc` Starting from this point, we essentially replicate the workings of `\advice@handle`, adapted to L^AT_EX environments.

```
2585 \def\advice@begin@env@rc#1#2#3{%
```

We first load the stage-one configuration for environment `#3` in namespace `#1`.

```
2586 \advice@init@i{#1}{#3}%
```

This defined `\AdviceRunConditions` for the environment. We can now check its run conditions. If they are not satisfied, we bail out by executing the environment's bailout handler followed by the original `\begin` (`\advice@original@cs{#1}{#2}`) plus the environment name (`#3`).

```
2587 \AdviceRunConditions
2588 \ifAdviceRun
2589 \expandafter\advice@begin@env@outer
2590 \else
2591 \AdviceBailoutHandler
2592 \expandafter\advice@original@cs
2593 \fi
2594 {#1}{#2}{#3}%
2595 }
```

`\advice@begin@env@outer` We define the macros expected by the outer handler, see `\advice@handle@outer`, load the second-stage configuration, and execute the environment's outer handler.

```
2596 \def\advice@begin@env@outer#1#2#3{%
2597 \def\AdviceNamespace{#1}%
2598 \def\AdviceName{#3}%
2599 \let\AdviceCsname\advice@undefined
2600 \def\AdviceReplaced{#2{#3}}%
2601 \def\AdviceOriginal{\AdviceGetOriginal{#1}{#2}{#3}}%
2602 \advice@init@I{#1}{#3}%
2603 \AdviceOuterHandler
2604 }
2605 </latex>
```

8.1.5 Error messages

Define error messages for the entire package. Note that `\advice@(de)activate@error@...` implement try activate.

```
2606 \def\advice@activate@error@activated#1#2#3#4{%
2607 \ifadvice@activate@try
2608 \else
2609 \PackageError{advice (#1)}{#3 "\string#2" is #4 activated}{}%
2610 \fi
2611 }
2612 \def\advice@activate@error@undefined#1#2#3#4{%
2613 \ifadvice@activate@try
2614 \else
2615 \PackageError{advice (#1)}{
2616 #3 "\string#2" you are trying to #4activate is not defined}{}%
2617 \fi
2618 }
2619 \def\advice@deactivate@error@changed#1#2{%
2620 \ifadvice@activate@try
2621 \else
2622 \PackageError{advice (#1)}{The definition of "\string#2" has changed since we
2623 have activated it. Has somebody overridden our command?}{If you have tried
2624 to deactivate so that you could immediately reactivate, you may want to try
2625 "force activate".}%
2626 \fi
```

```

2627 }
2628 \def\advice@error@advice@notcs#1#2{%
2629   \PackageError{advice}{The first argument of key "#1" should be either a single
2630     control sequence or an environment name, not "#2"}{}%
2631 }
2632 \def\advice@error@activate@notcsorenv#1#2{%
2633   \PackageError{advice}{Each item in the value of key "#1activate" should be
2634     either a control sequence or an environment name, not "#2".}{}%
2635 }
2636 \def\advice@error@storecs@notcs#1#2{%
2637   \PackageError{advice}{The value of key "#1" should be a single control sequence,
2638     not "\string#2"}{}%
2639 }
2640 \def\advice@error@noinnerhandler#1{%
2641   \PackageError{advice (\AdviceNamespace)}{The inner handler for
2642     "\expandafter\string\AdviceName" is not defined}{}%
2643 }

```

8.1.6 Tracing

We implement tracing by adding the tracing information to the handlers after we load them. So it is the handlers themselves which, if and when they are executed, will print out that this is happening.

`\AdviceTracingOn` Enable and disable tracing.

`\AdviceTracingOff`

```

2644 \def\AdviceTracingOn{%
2645   \let\advice@init@i\advice@trace@init@i
2646   \let\advice@init@I\advice@trace@init@I
2647 }
2648 \def\AdviceTracingOff{%
2649   \let\advice@init@i\advice@setup@init@i
2650   \let\advice@init@I\advice@setup@init@I
2651 }

```

`\advice@typeout` The tracing output routine; the typeout macro depends on the format. In \LaTeX , we use stream

`\advice@trace` `\@unused`, which is guaranteed to be unopened, so that the output will go to the terminal and the log. \ConTeXt , we don't muck about with write streams but simply use Lua function `texio.write_nl`. In plain \TeX , we use either Lua or the stream, depending on the engine; we use a high stream number 128 although the good old 16 would probably work just as well.

```

2652 <plain> \ifdefined\luatexversion
2653 <!latex> \long\def\advice@typeout#1{\directlua{texio.write_nl("\luaescapestring{#1}")}}
2654 <plain> \else
2655 <latex> \def\advice@typeout{\immediate\write\@unused}
2656 <plain> \def\advice@typeout{\immediate\write128}
2657 <plain> \fi
2658 \def\advice@trace#1{\advice@typeout{[tracing advice] #1}}

```

`\advice@trace@init@i` Install the tracing code.

`\advice@trace@init@I`

```

2659 \def\advice@trace@init@i#1#2{%
2660   \advice@trace{Advising \detokenize\expandafter{\string#2} (\detokenize{#1})}%
2661   \advice@trace{\space\space Original command meaning:
2662     \expandafter\expandafter\expandafter\meaning\advice@original@cs{#1}{#2}}%
2663   \advice@setup@init@i{#1}{#2}%
2664   \edef\AdviceRunConditions{%

```

We first execute the original run conditions, so that we can show the result.

```

2665     \expandonce\AdviceRunConditions
2666     \noexpand\advice@trace{\space\space
2667       Executing run conditions:

```

```

2668     \detokenize\expandafter{\AdviceRunConditions}
2669     -->
2670     \noexpand\ifAdviceRun true\noexpand\else false\noexpand\fi
2671   }%
2672 }%
2673 \edef\AdviceBailoutHandler{%
2674   \noexpand\advice@trace{\space\space
2675     Executing bailout handler:
2676     \detokenize\expandafter{\AdviceBailoutHandler}}}%
2677   \expandonce\AdviceBailoutHandler
2678 }%
2679 }
2680 \def\advice@trace@init@I#1#2{%
2681   \advice@setup@init@I{#1}{#2}%
2682   \edef\AdviceOuterHandler{%
2683     \noexpand\advice@trace{\space\space
2684       Executing outer handler:
2685       \detokenize\expandafter{\AdviceOuterHandler}}}%
2686     \expandonce\AdviceOuterHandler
2687   }%
2688   \edef\AdviceCollector{%
2689     \noexpand\advice@trace{\space\space
2690       Executing collector:
2691       \detokenize\expandafter{\AdviceCollector}}}%
2692     \noexpand\advice@trace{\space\space\space\space
2693       Argument specification:
2694       \detokenize\expandafter{\AdviceArgs}}}%
2695     \noexpand\advice@trace{\space\space\space\space
2696       Options:
2697       \detokenize\expandafter{\AdviceCollectorOptions}}}%
2698     \noexpand\advice@trace{\space\space\space\space
2699       Raw options:
2700       \detokenize\expandafter{\AdviceRawCollectorOptions}}}%

```

Collargs' return complicates tracing of the received argument. We put the code for remembering its value among the raw collector options. The default is 0; it is needed when we're using a collector other than `\CollectArguments`, the assumption being that external collectors will always return the collected arguments braced.

```

2701   \unexpanded{%
2702     \gdef\advice@collargs@return{0}%
2703     \appto\AdviceRawCollectorOptions{\advice@remember@collargs@return}%
2704   }%
2705   \expandonce\AdviceCollector
2706 }%
2707 \edef\advice@inner@handler@trace@do{%
2708   \noexpand\advice@trace{\space\space
2709     Executing inner handler:
2710     \detokenize\expandafter{\AdviceInnerHandler}}}%

```

When this macro is executed, the received arguments are waiting for us in `\toks0`.

```

2711   \noexpand\advice@trace{\space\space\space\space
2712     Received arguments\noexpand\advice@inner@handler@trace@printcollargsreturn:
2713     \noexpand\detokenize\noexpand\expandafter{\unexpanded{\the\toks0}}}%
2714   \noexpand\advice@trace{\space\space\space\space
2715     Options:
2716     \detokenize\expandafter{\AdviceOptions}}}%
2717   \expandonce{\AdviceInnerHandler}%
2718 }%
2719 \def\AdviceInnerHandler{\advice@inner@handler@trace}%
2720 }
2721 \def\advice@remember@collargs@return{%

```



```

2722 \global\let\advice@collargs@return\collargsReturn
2723 }

```

This is the entry point into the tracing inner handler. It will either get the received arguments as a braced argument (when Collargs' `return=0`), or from `\collargsArgs` otherwise. We don't simply always inspect `\collargsArgs` because foreign argument collectors will not use this token register; the assumption is that they will always return the collected arguments braced.

```

2724 \def\advice@inner@handler@trace{%
2725   \ifnum\advice@collargs@return=0
2726     \expandafter\advice@inner@handler@trace@i
2727   \else
2728     \expandafter\advice@inner@handler@trace@ii
2729   \fi
2730 }
2731 \def\advice@inner@handler@trace@i#1{%
2732   \toks0={#1}%
2733   \advice@inner@handler@trace@do{#1}%
2734 }
2735 \def\advice@inner@handler@trace@ii{%
2736   \expandafter\toks\expandafter0\expandafter{\the\collargsArgs}%
2737   \advice@inner@handler@trace@do
2738 }
2739 \def\advice@inner@handler@trace@printcollargsreturn{%
2740   \ifnum\advice@collargs@return=0
2741   \else
2742     \space(collargs return=%
2743     \ifcase\advice@collargs@return braced\or plain\or no\fi
2744     )%
2745   \fi
2746 }

```

```

2747 <plain>\resetatcatcode
2748 <context>\stopmodule
2749 <context>\protect
2750 </main>

```

8.1.7 The TikZ collector

In this section, we implement the argument collector for command `\tikz`, which has idiosyncratic syntax, see §12.2.2 of the [TikZ & PGF manual](#):

- `\tikz<animation spec>[<options>]{<picture code>}`
- `\tikz<animation spec>[<options>]<picture command>;`

where `<animation spec> = (:<key>={<value>})*`.

The TikZ code resides in a special file. It is meant to be `\input` at any time, so we need to temporarily assign @ category code 11.

```

2751 <*tikz>
2752 \edef\advice@resetatcatcode{\catcode`\noexpand@\the\catcode`\@relax}%
2753 \catcode`\@=11
2754 \def\AdviceCollectTikZArguments{%

```

We initialize the token register which will hold the collected arguments, and start the collection. Nothing of note happens until ...

```

2755   \toks0={}%
2756   \advice@tikz@anim
2757 }
2758 \def\advice@tikz@anim{%
2759   \pgfutil@ifnextchar[{\advice@tikz@opt}{%
2760     \pgfutil@ifnextchar:{\advice@tikz@anim@a}{%
2761       \advice@tikz@code}}%]

```



```

2762 }
2763 \def\advice@tikz@anim@a#1=#2{%
2764   \toksapp0{#1={#2}}%
2765   \advice@tikz@anim
2766 }
2767 \def\advice@tikz@opt[#1]{%
2768   \toksapp0{[#1]}%
2769   \advice@tikz@code
2770 }
2771 \def\advice@tikz@code{%
2772   \pgfutil@ifnextchar\bgroup\advice@tikz@braced\advice@tikz@single
2773 }
2774 \long\def\advice@tikz@braced#1{\toksapp0{#1}}\advice@tikz@done}
2775 \def\advice@tikz@single#1;\toksapp0{#1;}\advice@tikz@done}

```

... we finish collecting the arguments, when we execute the inner handler, with the (braced) collected arguments is its sole argument.

```

2776 \def\advice@tikz@done{%
2777   \expandafter\AdviceInnerHandler\expandafter{\the\toks0}%
2778 }
2779 \adviceresetatcatcode
2780 </tikz>

```

Local Variables: TeX-engine: luatex TeX-master: "doc/memoize-code.tex" TeX-auto-save: nil End:

8.2 Argument collection with CollArgs

Package CollArgs provides commands `\CollectArguments` and `\CollectArgumentsRaw`, which (what a surprise!) collect the arguments conforming to the given (slightly extended) `xparse` argument specification. The package was developed to help out with automemoization (see section 5). It started out as a few lines of code, but had grown once I realized I want automemoization to work for verbatim environments as well — the environment-collecting code is based on Bruno Le Floch’s package `cprotect` — and had then grown some more once I decided to support the `xparse` argument specification in full detail, and to make the verbatim mode flexible enough to deal with a variety of situations.

The implementation of this package does not depend on `xparse`. Perhaps this is a mistake, especially as the `xparse` code is now included in the base L^AT_EX, but the idea was to have a light-weight package (not sure this is the case anymore, given all the bells and whistles), to have its functionality available in plain T_EX and ConT_EXt as well (same as Memoize), and, perhaps most importantly, to have the ability to collect the arguments verbatim.

Identification

```

2781 <latex>\ProvidesPackage{collargs}[2024/01/02 v1.1.0 Collect arguments of any command]
2782 <context>%D \module[
2783 <context>%D   file=t-collargs.tex,
2784 <context>%D   version=1.1.0,
2785 <context>%D   title=CollArgs,
2786 <context>%D   subtitle=Collect arguments of any command,
2787 <context>%D   author=Saso Zivanovic,
2788 <context>%D   date=2024-01-02,
2789 <context>%D   copyright=Saso Zivanovic,
2790 <context>%D   license=LPPL,
2791 <context>%D ]
2792 <context>\writestatus{loading}{ConTeXt User Module / collargs}
2793 <context>\unprotect
2794 <context>\startmodule[collargs]

```

Required packages

```

2795 <latex>\RequirePackage{pgfkeys}

```

```

2796 <plain>\input pgfkeys
2797 <context>\input t-pgfkey
2798 <latex>\RequirePackage{etoolbox}
2799 <plain,context>\input etoolbox-generic
2800 <plain>\edef\resetatcatcode{\catcode`\noexpand\@the\catcode`\@the\relax}
2801 <plain>\catcode`\@11\relax

```

\toksapp Macros for appending to a token register. We don't have to define them in LuaTeX, where they exist as primitives. Same as these primitives, our macros accept either a register number or a **\etoksapp** **\toksdeffed** control sequence as the (unbraced) #1; #2 is the text to append.

```

<xtoksapp>
2802 \ifdefined\luatexversion
2803 \else
2804   \def\toksapp{\toks@cs@or@num\@toksapp}
2805   \def\gtoksapp{\toks@cs@or@num\@gtoksapp}
2806   \def\etoksapp{\toks@cs@or@num\@etoksapp}
2807   \def\xtoksapp{\toks@cs@or@num\@xtoksapp}
2808   \def\toks@cs@or@num#1#2#{%

```

Test whether #2 (the original #1) is a number or a control sequence.

```

2809     \ifnum-2>-1#2

```

It is a number. **\toks@cs@or@num@num** will gobble **\toks@cs@or@num@cs** below.

```

2810     \expandafter\toks@cs@or@num@num

```

The register control sequence in #2 is skipped over in the false branch.

```

2811     \fi
2812     \toks@cs@or@num@cs{#1}{#2}%
2813 }

```

#1 is one of **\@toksapp** and friends. The second macro prefixes the register number by **\toks**.

```

2814   \def\toks@cs@or@num@cs#1#2{#1{#2}}
2815   \def\toks@cs@or@num@num\toks@cs@or@num@cs#1#2{#1{\toks#2 }}

```

Having either **\tokscs** or **\toks<number>** in #1, we can finally do the real job.

```

2816   \long\def\@toksapp#1#2{#1\expandafter{\the#1#2}}%
2817   \long\def\@etoksapp#1#2{#1\expandafter{\expanded{\the#1#2}}}%
2818   \long\def\@gtoksapp#1#2{\global#1\expandafter{\the#1#2}}%
2819   \long\def\@xtoksapp#1#2{\global#1\expandafter{\expanded{\the#1#2}}}%
2820 \fi

```

\CollectArguments **\CollectArguments** takes three arguments: the optional #1 is the option list, processed **\CollectArgumentsRaw** by **pgfkeys** (given the grouping structure, these options will apply to all arguments); the mandatory #2 is the **xparse**-style argument specification; the mandatory #3 is the “next” command (or a sequence of commands). The argument list is expected to start immediately after the final argument; **\CollectArguments** parses it, effectively figuring out its extent, and then passes the entire argument list to the “next” command (as a single argument).

\CollectArgumentsRaw differs only in how it takes and processes the options. For one, these should be given as a mandatory argument. Furthermore, they do not take the form of a keylist, but should deploy the “programmer’s interface.” #1 should thus be a sequence of invocations of the macro counterparts of the keys defined in section 8.2.1, which can be recognized as starting with **\collargs** followed by a capital letter, e.g. **\collargsCaller**. Note that **\collargsSet** may also be used in #1. (The “optional,” i.e. bracketed, argument of **\CollectArgumentsRaw** is in fact mandatory.)

```

2821 \protected\def\CollectArguments{%
2822   \pgf@keys@utilifnextchar[\CollectArguments@i{\CollectArgumentsRaw{}}}%
2823 }
2824 \def\CollectArguments@i[#1]{\CollectArgumentsRaw{\collargsSet{#1}}
2825 \protected\def\CollectArgumentsRaw#1#2#3{%

```

This group will be closed by `\collargs@.` once we grinded through the argument specification.

```
2826 \beginingroup
```

Initialize category code fixing; see section 8.2.6 for details. We have to do this before applying the settings, so that `\collargsFixFromNoVerbatim` et al can take effect.

```
2827 \global\let\ifcollargs@last@verbatim\ifcollargs@verbatim
2828 \global\let\ifcollargs@last@verbatimbraces\ifcollargs@verbatimbraces
2829 \global\collargs@double@fixfalse
```

Apply the settings.

```
2830 \collargs@verbatim@wrap{#1}%
```

Initialize the space-grabber.

```
2831 \collargs@init@grabspaces
```

Remember the code to execute after collection.

```
2832 \def\collargs@next{#3}%
```

Initialize the token register holding the collected arguments.

```
2833 \ifcollargsClearArgs
2834   \global\collargsArgs{}%
2835 \fi
```

Execute the central loop macro, which expects the argument specification #2 to be delimited from the following argument tokens by a dot.

```
2836 \collargs@#2.%
2837 }
```

`\collargsSet` This macro processes the given keys in the `/collargs` keypath. When it is used to process options given by the end user (the optional argument to `\CollectArguments`, and the options given within the argument specification, using the new modifier `&`), its invocation should be wrapped in `\collargs@verbatim@wrap` to correctly deal with the changes of the verbatim mode.

```
2838 \def\collargsSet#1{\pgfqkeys{/collargs}{#1}}
```

8.2.1 The keys

`\collargs@cs@cases` If the first argument of this auxiliary macro is a single control sequence, then the second argument is executed. If the first argument starts with a control sequence but this control sequence does not form the entire argument, the third argument is executed. Otherwise, the fourth argument is executed.

This macro is defined in package `CollArgs` because we use it in key `caller` below, but it is really useful in package `Auto`, where having it we don't have to bother the end-user with a separate keys for commands and environments, but automatically detect whether the argument of `auto` and `(de)activate` is a command or an environment.

```
2839 \def\collargs@cs@cases#1{\collargs@cs@cases@i#1\collargs@cs@cases@end}
2840 \let\collargs@cs@cases@end\relax
2841 \def\collargs@cs@cases@i{\futurelet\collargs@temp\collargs@cs@cases@ii}
2842 \def\collargs@cs@cases@ii#1#2\collargs@cs@cases@end{%
2843   \ifcat\noexpand\collargs@temp\relax
2844     \ifx\relax#2\relax
2845       \expandafter\expandafter\expandafter\@firstofthree
2846     \else
2847       \expandafter\expandafter\expandafter\@secondofthree
2848     \fi
```

```

2849 \else
2850 \expandafter\@thirdofthree
2851 \fi
2852 }
2853 \def\@firstofthree#1#2#3{#1}
2854 \def\@secondofthree#1#2#3{#2}
2855 \def\@thirdofthree#1#2#3{#3}

```

caller Every macro which grabs a part of the argument list will be accessed through the “caller” control sequence, so that \TeX ’s reports of any errors in the argument structure can contain a command name familiar to the author.⁴ For example, if the argument list “originally” belonged to command \foo with argument structure \r() , but no parentheses follow in the input, we want \TeX to complain that `Use of \foo doesn't match its definition`. This can be achieved by setting $\text{\caller}=\text{\foo}$; the default is $\text{\caller}=\text{\CollectArguments}$, which is still better than seeing an error involving some random internal control sequence. It is also ok to set an environment name as the caller, see below.

The key and macro defined below store the caller control sequence into \collargs@caller , e.g. when we say $\text{\caller}=\text{\foo}$, we effectively execute $\text{\def\collargs@caller\foo}$.

```

2856 \collargsSet{
2857   caller/.code={\collargsCaller{#1}},
2858 }
2859 \def\collargsCaller#1{%
2860   \collargs@cs@cases{#1}{%
2861     \let\collargs@temp\collargs@caller@cs
2862   }{%
2863     \let\collargs@temp\collargs@caller@csandmore
2864   }{%
2865     \let\collargs@temp\collargs@caller@env
2866   }%
2867   \collargs@temp{#1}%
2868 }
2869 \def\collargs@caller@cs#1{%

```

If \#1 is a single control sequence, just use that as the caller.

```

2870   \def\collargs@caller{#1}%
2871 }
2872 \def\collargs@caller@csandmore#1{%

```

If \#1 starts with a control sequence, we don’t complain, but convert the entire \#1 into a control sequence.

```

2873   \begingroup
2874   \escapechar -1
2875   \expandafter\endgroup
2876   \expandafter\def\expandafter\collargs@caller\expandafter{%
2877     \csname\string#1\endcsname
2878   }%
2879 }
2880 \def\collargs@caller@env#1{%

```

If \#1 does not start with a control sequence, we assume that is an environment name, so we prepend `start` in \ConTeXt , and dress it up into $\text{\begin\{#1}}$ in \LaTeX .

```

2881   \expandafter\def\expandafter\collargs@caller\expandafter{%
2882     \csname
2883 <context>    start%
2884 <latex>    begin{%
2885             #1%
2886 <latex>    }%

```

⁴The idea is borrowed from package `environ`, which is in turn based on code from `amsmath`.

```

2887 \endcsname
2888 }%
2889 }
2890 \collargsCaller\CollectArguments

```

`\ifcollargs@verbatim` The first of these conditional signals that we're collecting the arguments in one of the `\ifcollargs@verbatimbraces` verbatim modes; the second one signals the `verb` mode in particular.

```

2891 \newif\ifcollargs@verbatim
2892 \newif\ifcollargs@verbatimbraces

```

`verbatim` These keys set the verbatim mode macro which will be executed by `\collargsSet` after `verb` processing all keys. The verbatim mode macros `\collargsVerbatim`, `\collargsVerb` `no verbatim` and `\collargsNoVerbatim` are somewhat complex; we postpone their definition until section 8.2.5. Their main effect is to set conditionals `\ifcollargs@verbatim` and `\ifcollargs@verbatimbraces`, which are inspected by the argument type handlers — and to make the requested category code changes, of course.

Here, note that the verbatim-selection code is not executed while the keylist is being processed. Rather, the verbatim keys simply set the macro which will be executed *after* the keylist is processed, and this is why processing of a keylist given by the user must be always wrapped in `\collargs@verbatim@wrap`.

```

2893 \collargsSet{
2894   verbatim/.code={\let\collargs@apply@verbatim\collargsVerbatim},
2895   verb/.code={\let\collargs@apply@verbatim\collargsVerb},
2896   no verbatim/.code={\let\collargs@apply@verbatim\collargsNoVerbatim},
2897 }
2898 \def\collargs@verbatim@wrap#1{%
2899   \let\collargs@apply@verbatim\relax
2900   #1%
2901   \collargs@apply@verbatim
2902 }

```

`fix from verbatim` These keys and macros should be used to request a category code fix, when the offending `fix from verb` tokenization took place prior to invoking `\CollectArguments`; see section 8.2.6 for details. While I assume that only `\collargsFixFromNoVerbatim` will ever be used `\collargsFixFromVerbatim` (and it is used by `\mmz`), we provide macros for all three transitions, for completeness.

```

\collargsFixFromVerb
\collargsFixFromNoVerbatim
2903 \collargsSet{
2904   fix from verbatim/.code={\collargsFixFromVerbatim},
2905   fix from verb/.code={\collargsFixFromVerb},
2906   fix from no verbatim/.code={\collargsFixFromNoVerbatim},
2907 }
2908 \def\collargsFixFromNoVerbatim{%
2909   \global\collargs@fix@requestedtrue
2910   \global\let\ifcollargs@last@verbatim\iffalse
2911 }
2912 \def\collargsFixFromVerbatim{%
2913   \global\collargs@fix@requestedtrue
2914   \global\let\ifcollargs@last@verbatim\iftrue
2915   \global\let\ifcollargs@last@verbatimbraces\iftrue
2916 }
2917 \def\collargsFixFromVerb{%
2918   \global\collargs@fix@requestedtrue
2919   \global\let\ifcollargs@last@verbatim\iftrue
2920   \global\let\ifcollargs@last@verbatimbraces\iffalse
2921 }

```

`braces` Set the characters which are used as the grouping characters in the full verbatim mode. The user is only required to do this when multiple character pairs serve as the grouping characters. The underlying macro, `\collargsBraces`, will be defined in section 8.2.5.

```

2922 \collargsSet{
2923   braces/.code={\collargsBraces{#1}}%
2924 }

```

environment Set the environment name.

```

\collargsEnvironment
2925 \collargsSet{
2926   environment/.estore in=\collargs@b@envname
2927 }
2928 \def\collargsEnvironment#1{\edef\collargs@b@envname{#1}}
2929 \collargsEnvironment{}

```

begin tag When **begin tag**/**end tag** is in effect, the **begin**/**end-tag** will be prepended/appended to the environment body. **tags** is a shortcut for setting **begin tag** and **end tag** simultaneously.

```

\ifcollargsBeginTag
\ifcollargsEndTag
\ifcollargsAddTags
2930 \collargsSet{
2931   begin tag/.is if=collargsBeginTag,
2932   end tag/.is if=collargsEndTag,
2933   tags/.style={begin tag=#1, end tag=#1},
2934   tags/.default=true,
2935 }
2936 \newif\ifcollargsBeginTag
2937 \newif\ifcollargsEndTag

```

ignore nesting When this key is in effect, we will ignore any **\begin{<name>}**s and simply grab everything up to the first **\end{<name>}** (again, the markers are automatically adapted to the format).

```

2938 \collargsSet{
2939   ignore nesting/.is if=collargsIgnoreNesting,
2940 }
2941 \newif\ifcollargsIgnoreNesting

```

ignore other tags This key is only relevant in the non-verbatim and partial verbatim modes in L^AT_EX.

\ifcollargsIgnoreOtherTags When it is in effect, CollArgs checks the environment name following each **\begin** and **\end**, ignoring the tags with an environment name other than **\collargs@b@envname**.

```

2942 \collargsSet{
2943   ignore other tags/.is if=collargsIgnoreOtherTags,
2944 }
2945 \newif\ifcollargsIgnoreOtherTags

```

(append/prepend) (pre/post)processor These keys and macros populate the list of preprocessors,

\collargs(Append/Prepend)(Pre/Post)processor **\collargs@preprocess@arg**, and the list of post-processors, **\collargs@postprocess@arg**, executed in **\collargs@appendarg**.

```

2946 \collargsSet{
2947   append preprocessor/.code={\collargsAppendPreprocessor{#1}},
2948   prepend preprocessor/.code={\collargsPrependPreprocessor{#1}},
2949   append postprocessor/.code={\collargsAppendPostprocessor{#1}},
2950   prepend postprocessor/.code={\collargsPrependPostprocessor{#1}},
2951 }
2952 \def\collargsAppendPreprocessor#1{\appto\collargs@preprocess@arg{#1}}
2953 \def\collargsPrependPreprocessor#1{\preto\collargs@preprocess@arg{#1}}
2954 \def\collargsAppendPostprocessor#1{\appto\collargs@postprocess@arg{#1}}
2955 \def\collargsPrependPostprocessor#1{\preto\collargs@postprocess@arg{#1}}

```

clear (pre/post)processors These keys and macros clear the pre- and post-processor lists, which are

\collargsClear(Pre/Post)processors initially empty as well.

```

2956 \def\collargs@preprocess@arg{}

```



```

2957 \def\collargs@postprocess@arg{}
2958 \collargsSet{
2959   clear preprocessors/.code={\collargsClearPreprocessors},
2960   clear postprocessors/.code={\collargsClearPostprocessors},
2961 }
2962 \def\collargsClearPreprocessors{\def\collargs@preprocess@arg{}}%
2963 \def\collargsClearPostprocessors{\def\collargs@postprocess@arg{}}%

```

(append/prepend) expandable (pre/post)processor These keys and macros simplify the definition of expandable processors. Note that expandable processors are added to the same list as non-expandable processors.

```

2964 \collargsSet{
2965   append expandable preprocessor/.code={\collargsAppendExpandablePreprocessor{#1}},
2966   prepend expandable preprocessor/.code={\collargsPrependExpandablePreprocessor{#1}},
2967   append expandable postprocessor/.code={\collargsAppendExpandablePostprocessor{#1}},
2968   prepend expandable postprocessor/.code={\collargsPrependExpandablePostprocessor{#1}},
2969 }
2970 \def\collargsAppendExpandablePreprocessor#1{%
2971   \appto\collargs@preprocess@arg{%
2972     \collargsArg\expandafter{\expanded{#1}}}%
2973 }%
2974 }
2975 \def\collargsPrependExpandablePreprocessor#1{%
2976   \preto\collargs@preprocess@arg{%
2977     \collargsArg\expandafter{\expanded{#1}}}%
2978 }%
2979 }
2980 \def\collargsAppendExpandablePostprocessor#1{%
2981   \appto\collargs@postprocess@arg{%
2982     \collargsArg\expandafter{\expanded{#1}}}%
2983 }%
2984 }
2985 \def\collargsPrependExpandablePostprocessor#1{%
2986   \preto\collargs@postprocess@arg{%
2987     \collargsArg\expandafter{\expanded{#1}}}%
2988 }%
2989 }

```

no delimiters When this conditional is in effect, the delimiter wrappers set by `\collargs@wrap` are `\ifcollargsNoDelimiters` ignored by `\collargs@appendarg`.

```

2990 \collargsSet{%
2991   no delimiters/.is if=collargsNoDelimiters,
2992 }
2993 \newif\ifcollargsNoDelimiters

```

clear args When this conditional is set to false, the global token register `\collargsArgs` receiving `\ifcollargsClearArgs` the collected arguments is not cleared prior to argument collection.

```

2994 \collargsSet{%
2995   clear args/.is if=collargsClearArgs,
2996 }
2997 \newif\ifcollargsClearArgs
2998 \collargsClearArgstrue

```

return Exiting `\CollectArguments`, should the next-command be followed by the braced collected arguments, collected arguments as they are, or nothing?

```

2999 \collargsSet{%
3000   return/.is choice,
3001   return/braced/.code=\collargsReturnBraced,

```

```

3002 return/plain/.code=\collargsReturnPlain,
3003 return/no/.code=\collargsReturnNo,
3004 }
3005 \def\collargsReturnBraced{\def\collargsReturn{0}}
3006 \def\collargsReturnPlain{\def\collargsReturn{1}}
3007 \def\collargsReturnNo{\def\collargsReturn{2}}
3008 \collargsReturnBraced

```

```

alias
\collargsAlias
3009 \collargsSet{%
3010 alias/.code 2 args=\collargsAlias{#1}{#2}%
3011 }
3012 \def\collargsAlias#1#2{%
3013 \csdef{collargs@#1}{\collargs@@@#2}%
3014 }

```

8.2.2 The central loop

The central loop is where we grab the next *<token>* from the argument specification and execute the corresponding argument type or modifier handler, `\collargs@<token>`. The central loop consumes the argument type *<token>*; the handler will see the remainder of the argument specification (which starts with the arguments to the argument type, if any, e.g. by `()` of `d()`), followed by a dot, and then the tokens list from which the arguments are to be collected. It is the responsibility of handler to preserve the rest of the argument specification and reexecute the central loop once it is finished.

`\collargs@` Each argument is processed in a group to allow for local settings. This group is closed by `\collargs@appendarg`.

```

3015 \def\collargs@{%
3016 \begingroup
3017 \collargs@@@
3018 }

```

`\collargs@@@` This macro is where modifier handlers reenter the central loop — we don't want modifiers to open a group, because their settings should remain in effect until the next argument. Furthermore, modifiers do not trigger category code fixes.

```

3019 \def\collargs@@@#1{%
3020 \collargs@in@{#1}{&+!>.|}%
3021 \ifcollargs@in@
3022 \expandafter\collargs@@@iii
3023 \else
3024 \expandafter\collargs@@@i
3025 \fi
3026 #1%
3027 }
3028 \def\collargs@@@i#1.{%

```

Fix the category code of the next argument token, if necessary, and then proceed with the main loop.

```

3029 \collargs@fix{\collargs@@@i#1.}%
3030 }

```

Reset the fix request and set the last verbatim conditionals to the current state.

```

3031 \def\collargs@@@iii{%
3032 \global\collargs@fix@requestedfalse
3033 \global\let\ifcollargs@last@verbatim\ifcollargs@verbatim
3034 \global\let\ifcollargs@last@verbatimbraces\ifcollargs@verbatimbraces
3035 \collargs@@@iii
3036 }

```


Call the modifier or argument type handler denoted by the first token of the remainder of the argument specification.

```
3037 \def\collargs@@@iii#1{%
3038   \ifcsname collargs@#1\endcsname
3039     \csname collargs@#1\expandafter\endcsname
3040   \else
```

We throw an error if the token refers to no argument type or modifier.

```
3041     \collargs@error@badtype{#1}%
3042   \fi
3043 }
```

Throwing an error stops the processing of the argument specification, and closes the group opened in `\collargs@i`.

```
3044 \def\collargs@error@badtype#1#2.{%
3045   \PackageError{collargs}{Unknown xparse argument type or modifier "#1"
3046     for "\expandafter\string\collargs@caller\space"}{}%
3047   \endgroup
3048 }
```

`\collargs@&` We extend the `xparse` syntax with modifier `&`, which applies the given options to the following (and only the following) argument. If `&` is followed by another `&`, the options are expected to occur in the raw format, like the options given to `\CollectArgumentsRaw`. Otherwise, the options should take the form of a keylist, which will be processed by `\collargsSet`. In any case, the options should be given within the argument specification, immediately following the (single or double) `&`.

```
3049 \csdef{collargs@&}{%
3050   \futurelet\collargs@temp\collargs@amp@i
3051 }
3052 \def\collargs@amp@i{%
```

In `ConTeXt`, `&` has character code “other” in the text.

```
3053 <!context> \ifx\collargs@temp&%
3054 <context> \expandafter\ifx\detokenize{&}\collargs@temp
3055   \expandafter\collargs@amp@raw
3056   \else
3057     \expandafter\collargs@amp@set
3058   \fi
3059 }
3060 \def\collargs@amp@raw#1#2{%
3061   \collargs@verbatim@wrap{#2}%
3062   \collargs@@@
3063 }
3064 \def\collargs@amp@set#1{%
3065   \collargs@verbatim@wrap{\collargsSet{#1}}%
3066   \collargs@@@
3067 }
```

`\collargs@+` This modifier makes the next argument long, i.e. accept paragraph tokens.

```
3068 \csdef{collargs@+}{%
3069   \collargs@longtrue
3070   \collargs@@@
3071 }
3072 \newif\ifcollargs@long
```

`\collargs@>` We can simply ignore the processor modifier. (This, `xparse`'s processor, should not be confused with `CollArgs`'s processors, which are set using keys `append` `preprocessor` etc.)

```
3073 \csdef{collargs@>}#1{\collargs@@@}
```

`\collargs@!` Should we accept spaces before an optional argument following a mandatory argument (`xparse` manual, §1.1)? By default, yes. This modifier is only applicable to types `d` and `t`, and derived types, but, unlike `xparse`, we don't bother to enforce this; when used with other types, `!` simply has no effect.

```
3074 \csdef{collargs@!}{%
3075   \collargs@grabspacesfalse
3076   \collargs@@@
3077 }
```

`\collargsArgs` This token register is where we store the collected argument tokens. All assignments to this register are global, because it needs to survive the groups opened for individual arguments.

```
3078 \newtoks\collargsArgs
```

`\collargsArg` An auxiliary, but publicly available token register, used for processing the argument, and by some argument type handlers.

```
3079 \newtoks\collargsArg
```

`\collargs@.` This fake argument type is used to signal the end of the argument list. Note that this really counts as an extension of the `xparse` argument specification.

```
3080 \csdef{collargs@.}{%
```

Close the group opened in `\collargs@.`

```
3081   \endgroup
```

Close the main `\CollectArguments` group, fix the category code of the next token if necessary, and execute the next-code, followed by the collected arguments in braces. Any over-grabbed spaces are reinserted into the input stream, non-verbatim.

```
3082   \expanded{%
3083     \endgroup
3084     \noexpand\collargs@fix{%
3085       \expandonce\collargs@next
3086       \ifcase\collargsReturn\space
3087         {\the\collargsArgs}%
3088       \or
3089         \the\collargsArgs
3090       \fi
3091       \collargs@spaces
3092     }%
3093   }%
3094 }
```

8.2.3 Auxiliary macros

`\collargs@appendarg` This macro is used by the argument type handlers to append the collected argument to the storage (`\collargsArgs`).

```
3095 \long\def\collargs@appendarg#1{%
```

Temporarily store the collected argument into a token register. The processors will manipulate the contents of this register.

```
3096   \collargsArg={#1}%
```

This will clear the double-fix conditional, and potentially request a normal, single fix. We can do this here because this macro is only called when something is actually collected. For details, see section 8.2.6.

```
3097 \ifcollargs@double@fix
3098 \collargs@cancel@double@fix
3099 \fi
```

Process the argument with user-definable preprocessors, the wrapper defined by the argument type, and user-definable postprocessors.

```
3100 \collargs@preprocess@arg
3101 \ifcollargsNoDelimiters
3102 \else
3103 \collargs@process@arg
3104 \fi
3105 \collargs@postprocess@arg
```

Append the processed argument, preceded by any grabbed spaces (in the correct mode), to the storage.

```
3106 \xtoksapp\collargsArgs{\collargs@grabbed@spaces\the\collargsArg}%
```

Initialize the space-grabber.

```
3107 \collargs@init@grabspaces
```

Once the argument was appended to the list, we can close its group, opened by `\collargs@`.

```
3108 \endgroup
3109 }
```

`\collargs@wrap` This macro is used by argument type handlers to declare their delimiter wrap, like square brackets around the optional argument of type o. It uses `\collargs@addwrap`, defined in section 8.2.1, but adds to `\collargs@process@arg`, which holds the delimiter wrapper defined by the argument type handler. Note that this macro *appends* a wrapper, so multiple wrappers are allowed — this is used by type e handler.

```
3110 \def\collargs@wrap#1{%
3111 \appto\collargs@process@arg{%
3112 \long\def\collargs@temp##1{#1}%
3113 \expandafter\expandafter\expandafter\collargsArg
3114 \expandafter\expandafter\expandafter{%
3115 \expandafter\collargs@temp\expandafter{\the\collargsArg}%
3116 }%
3117 }%
3118 }
3119 \def\collargs@process@arg{}
```

`\collargs@defcollector` These macros streamline the usage of the “caller” control sequence. They are like a `\collargs@defusecollector` `\def`, but should not be given the control sequence to define, as they will automatically define the control sequence residing in `\collargs@caller`; the usage is thus `\collargs@defcollector<parameters>{<definition>}`. For example, if `\collargs@caller` holds `\foo`, `\collargs@defcollector#1{(#1)}` is equivalent to `\def\foo#1{(#1)}`. Macro `\collargs@defcollector` will only define the caller control sequence to be the collector, while `\collargs@defusecollector` will also immediately execute it.

```
3120 \def\collargs@defcollector#1#{%
3121 \ifcollargs@long\long\fi
3122 \expandafter\def\collargs@caller#1%
3123 }
3124 \def\collargs@defusecollector#1#{%
```

```

3125 \afterassignment\collargs@caller
3126 \ifcollargs@long\long\fi
3127 \expandafter\def\collargs@caller#1%
3128 }
3129 \def\collargs@letusecollector#1{%
3130 \expandafter\let\collargs@caller#1%
3131 \collargs@caller
3132 }
3133 \newif\ifcollargs@grabspaces
3134 \collargs@grabspacestrue

```

\collargs@init@grabspaces The space-grabber macro `\collargs@grabspaces` should be initialized by executing this macro. If `\collargs@grabspaces` is called twice without an intermediate initialization, it will assume it is in the same position in the input stream and simply bail out.

```

3135 \def\collargs@init@grabspaces{%
3136 \gdef\collargs@gs@state{0}%
3137 \gdef\collargs@spaces{}%
3138 \gdef\collargs@otherspaces{}%
3139 }

```

\collargs@grabspaces This auxiliary macro grabs any following spaces, and then executes the next-code given as the sole argument. The spaces will be stored into two macros, `\collargs@spaces` and `\collargs@otherspaces`, which store the spaces in the non-verbatim and the verbatim form. With the double storage, we can grab the spaces in the verbatim mode and use them non-verbatim, or vice versa. The macro takes a single argument, the code to execute after maybe grabbing the spaces.

```

3140 \def\collargs@grabspaces#1{%
3141 \edef\collargs@gs@next{\unexpanded{#1}}%
3142 \ifnum\collargs@gs@state=0
3143 \gdef\collargs@gs@state{1}%
3144 \expandafter\collargs@gs@i
3145 \else
3146 \expandafter\collargs@gs@next
3147 \fi
3148 }
3149 \def\collargs@gs@i{%
3150 \futurelet\collargs@temp\collargs@gs@g
3151 }

```

We check for grouping characters even in the verbatim mode, because we might be in the partial verbatim.

```

3152 \def\collargs@gs@g{%
3153 \ifcat\noexpand\collargs@temp\bgroup
3154 \expandafter\collargs@gs@next
3155 \else
3156 \ifcat\noexpand\collargs@temp\egroup
3157 \expandafter\expandafter\expandafter\collargs@gs@next
3158 \else
3159 \expandafter\expandafter\expandafter\collargs@gs@ii
3160 \fi
3161 \fi
3162 }
3163 \def\collargs@gs@ii{%
3164 \ifcollargs@verbatim
3165 \expandafter\collargs@gs@iii
3166 \else
3167 \expandafter\collargs@gs@iii
3168 \fi
3169 }

```

This works because the character code of a space token is always 32.

```

3170 \def\collargs@gs@iii{%
3171   \expandafter\ifx\space\collargs@temp
3172     \expandafter\collargs@gs@iv
3173   \else
3174     \expandafter\collargs@gs@next
3175   \fi
3176 }
3177 \expandafter\def\expandafter\collargs@gs@iv\space{%
3178   \gappto\collargs@spaces{ }%
3179   \xappto\collargs@otherspaces{\collargs@otherspace}%
3180   \collargs@gs@i
3181 }

```

We need the space of category 12 above.

```

3182 \begingroup\catcode\ =12\relax\gdef\collargs@otherspace{ }\endgroup
3183 \def\collargs@gs@iii#1{%

```

Macro `\collargs@cc` recalls the “outside” category code of character `#1`; see section 8.2.5.

```

3184   \ifnum\collargs@cc{#1}=10

```

We have a space.

```

3185     \expandafter\collargs@gs@iv
3186   \else
3187     \ifnum\collargs@cc{#1}=5

```

We have a newline.

```

3188     \expandafter\expandafter\expandafter\collargs@gs@v
3189   \else
3190     \expandafter\expandafter\expandafter\collargs@gs@next
3191   \fi
3192 \fi
3193 #1%
3194 }
3195 \def\collargs@gs@iv#1{%
3196   \gappto\collargs@otherspaces{#1}%

```

No matter how many verbatim spaces we collect, they equal a single non-verbatim space.

```

3197   \gdef\collargs@spaces{ }%
3198   \collargs@gs@i
3199 }
3200 \def\collargs@gs@v{%

```

Only add the first newline.

```

3201   \ifnum\collargs@gs@state=2
3202     \expandafter\collargs@gs@next
3203   \else
3204     \expandafter\collargs@gs@vi
3205   \fi
3206 }
3207 \def\collargs@gs@vi#1{%
3208   \gdef\collargs@gs@state{2}%
3209   \gappto\collargs@otherspaces{#1}%
3210   \gdef\collargs@spaces{ }%
3211   \collargs@gs@i
3212 }

```

`\collargs@maybegrabspace`s This macro grabs any following spaces, but it will do so only when conditional `\ifcollargs@grabspace`s, which can be *unset* by modifier `!`, is in effect. The macro is used by handlers for types `d` and `t`.

```

3213 \def\collargs@maybegrabspace{%
3214   \ifcollargs@grabspace
3215     \expandafter\collargs@grabspace
3216   \else
3217     \expandafter\@firstofone
3218   \fi
3219 }

```

`\collargs@grabbed@space`s This macro expands to either the verbatim or the non-verbatim variant of the grabbed spaces, depending on the verbatim mode in effect at the time of expansion.

```

3220 \def\collargs@grabbed@space{%
3221   \ifcollargs@verbatim
3222     \collargs@otherspace
3223   \else
3224     \collargs@space
3225   \fi
3226 }

```

`\collargs@reinsert@space`s Inserts the grabbed spaces back into the input stream, but with the category code appropriate for the verbatim mode then in effect. After the insertion, the space-grabber is initialized and the given next-code is executed in front of the inserted spaces.

```

3227 \def\collargs@reinsert@space#1{%
3228   \expanded{%
3229     \unexpanded{%
3230       \collargs@init@grabspace
3231       #1%
3232     }%
3233     \collargs@grabbed@space
3234   }%
3235 }

```

`\collargs@ifnextcat` An adaptation of `\pgf@keys@utilifnextchar` which checks whether the *category* code of the next non-space character matches the category code of `#1`.

```

3236 \long\def\collargs@ifnextcat#1#2#3{%
3237   \let\pgf@keys@utilreserved@d=#1%
3238   \def\pgf@keys@utilreserved@a{#2}%
3239   \def\pgf@keys@utilreserved@b{#3}%
3240   \futurelet\pgf@keys@utillet@token\collargs@ifncat}
3241 \def\collargs@ifncat{%
3242   \ifx\pgf@keys@utillet@token\pgf@keys@utilsp@token
3243     \let\pgf@keys@utilreserved@c\collargs@xifnch
3244   \else
3245     \ifcat\noexpand\pgf@keys@utillet@token\pgf@keys@utilreserved@d
3246       \let\pgf@keys@utilreserved@c\pgf@keys@utilreserved@a
3247     \else
3248       \let\pgf@keys@utilreserved@c\pgf@keys@utilreserved@b
3249     \fi
3250   \fi
3251   \pgf@keys@utilreserved@c}
3252 {%
3253   \def\:\{\collargs@xifncat}
3254   \expandafter\gdef\:\ {\futurelet\pgf@keys@utillet@token\collargs@ifncat}
3255 }

```

`\collargs@forrange` This macro executes macro `\collargs@do` for every integer from #1 and #2, both inclusive. `\collargs@do` should take a single parameter, the current number.

```

3256 \def\collargs@forrange#1#2{%
3257   \expanded{%
3258     \noexpand\collargs@forrange@i{\number#1}{\number#2}%
3259   }%
3260 }
3261 \def\collargs@forrange@i#1#2{%
3262   \ifnum#1>#2 %
3263     \expandafter\@gobble
3264   \else
3265     \expandafter\@firstofone
3266   \fi
3267   {%
3268     \collargs@do{#1}%
3269     \expandafter\collargs@forrange@i\expandafter{\number\numexpr#1+1\relax}{#2}%
3270   }%
3271 }

```

`\collargs@forranges` This macro executes macro `\collargs@do` for every integer falling into the ranges specified in #1. The ranges should be given as a comma-separated list of from-to items, e.g. 1-5,10-11.

```

3272 \def\collargs@forranges{\forcsvlist\collarg@forrange@i}
3273 \def\collarg@forrange@i#1{\collarg@forrange@ii#1-}
3274 \def\collarg@forrange@ii#1-#2-{\collargs@forrange{#1}{#2}}

```

`\collargs@percentchar` This macro holds the percent character of category 12.

```

3275 \begingroup
3276 \catcode`\%=12
3277 \gdef\collargs@percentchar{%}
3278 \endgroup

```

8.2.4 The handlers

`\collargs@l` We will first define the handler for the very funky argument type 1, which corresponds to TeX's `\def\foo#1#{...}`, which grabs (into #1) everything up to the first opening brace — not because this type is important or even recommended to use, but because the definition of the handler is very simple, at least for the non-verbatim case.

```

3279 \def\collargs@l#1.{%

```

Any pre-grabbed spaces in fact belong into the argument.

```

3280   \collargs@reinsert@spaces{\collargs@l@i#1.}%
3281 }
3282 \def\collargs@l@i{%

```

We request a correction of the category code of the delimiting brace if the verbatim mode changes for the next argument; for details, see section 8.2.6.

```

3283   \global\collargs@fix@requestedtrue

```

Most handlers will branch into the verbatim and the non-verbatim part using conditional `\ifcollargs@verbatim`. This handler is a bit special, because it needs to distinguish verbatim and non-verbatim *braces*, and braces are verbatim only in the full verbatim mode, i.e. when `\ifcollargs@verbatimbraces` is true.

```

3284   \ifcollargs@verbatimbraces
3285     \expandafter\collargs@l@verb
3286   \else
3287     \expandafter\collargs@l@ii
3288   \fi
3289 }

```

We grab the rest of the argument specification (`#1`), to be reinserted into the token stream when we reexecute the central loop.

```
3290 \def\collargs@l@ii#1.{%
```

In the non-verbatim mode, we merely have to define and execute the collector macro. The parameter text `##1##` (note the doubled hashes), which will put everything up to the first opening brace into the first argument, looks funky, but that's all.

```
3291 \collargs@defusecollector##1##{%
```

We append the collected argument, `##1`, to `\collargsArgs`, the token register holding the collected argument tokens.

```
3292 \collargs@appendarg{##1}%
```

Back to the central loop, with the rest of the argument specification reinserted.

```
3293 \collargs@#1.%
```

```
3294 }%
```

```
3295 }
```

```
3296 \def\collargs@l@verb#1.{%
```

In the verbatim branch, we need to grab everything up to the first opening brace of category code 12, so we want to define the collector with parameter text `##1{`, with the opening brace of category 12. We have stored this token in macro `\collargs@other@bgroup`, which we now need to expand.

```
3297 \expandafter\collargs@defusecollector
```

```
3298 \expandafter##\expandafter1\collargs@other@bgroup{%
```

Appending the argument works the same as in the non-verbatim case.

```
3299 \collargs@appendarg{##1}%
```

Reexecuting the central loop macro is a bit more involved, as we need to reinsert the verbatim opening brace (contrary to the regular brace above, the verbatim brace is consumed by the collector macro) back into the token stream, behind the reinserted argument specification.

```
3300 \expanded{%
```

```
3301 \noexpand\collargs@\unexpanded{#1.}%
```

```
3302 \collargs@other@bgroup
```

```
3303 }%
```

```
3304 }%
```

```
3305 }
```

`\collargs@u` Another weird type — `u⟨tokens⟩` reads everything up to the given `⟨tokens⟩`, i.e. this is \TeX 's `\def\foo#1⟨tokens⟩{...}` — but again, simple enough to allow us to showcase solutions to two recurring problems.

We start by branching into the verbatim mode (full or partial) or the non-verbatim mode.

```
3306 \def\collargs@u{%
```

```
3307 \ifcollargs@verbatim
```

```
3308 \expandafter\collargs@u@verb
```

```
3309 \else
```

```
3310 \expandafter\collargs@u@i
```

```
3311 \fi
```

```
3312 }
```

To deal with the verbatim mode, we only need to convert the above `⟨tokens⟩` (i.e. the argument of `u` in the argument specification) to category 12, i.e. we have to `\detokenize` them. Then, we may proceed as in the non-verbatim branch, `\collargs@u@ii`.

```
3313 \def\collargs@u@verb#1{%
```


The `\string` here is a temporary solution to a problem with spaces. Our verbatim mode has them of category “other”, but `\detokenize` produces a space of category “space” behind control words.

```
3314 \expandafter\collargs@u@i\expandafter{\detokenize\expandafter{\string#1}}%
3315 }
```

We then reinsert any pre-grabbed spaces into the stream, but we take care not to destroy the braces around our delimiter in the argument specification.

```
3316 \def\collargs@u@i#1#2.{%
3317 \collargs@reinsert@spaces{\collargs@u@ii{#1}#2.}%
3318 }
3319 \def\collargs@u@ii#1#2.{%
```

`#1` contains the delimiter tokens, so `##1` below will receive everything in the token stream up to these. But we have a problem: if we defined the collector as for the non-verbatim `1`, and the delimiter happened to be preceded by a single brace group, we would lose the braces. For example, if the delimiter was `-` and we received `{foo}-`, we would collect `foo-`. We solve this problem by inserting `\collargs@empty` (with an empty definition) into the input stream (at the end of this macro) — this way, the delimiter can never be preceded by a single brace group — and then expanding it away before appending to storage (within the argument of `\collargs@defusecollector`).

```
3320 \collargs@defusecollector##1#1{%
```

Define the wrapper which will add the delimiter tokens (`#1`) after the collected argument. The wrapper will be applied during argument processing in `\collargs@appendarg` (sandwiched between used-definable pre- and post-processors).

```
3321 \collargs@wrap{####1#1}%
```

Expand the first token in `##1`, which we know to be `\collargs@empty`, with empty expansion.

```
3322 \expandafter\collargs@appendarg\expandafter{##1}%
3323 \collargs@#2.%
3324 }%
```

Insert `\collargs@empty` into the input stream, in front of the “real” argument tokens.

```
3325 \collargs@empty
3326 }
3327 \def\collargs@empty{}
```

`\collargs@r` Finally, a real argument type: required delimited argument.

```
3328 \def\collargs@r{%
3329 \ifcollargs@verbatim
3330 \expandafter\collargs@r@verb
3331 \else
3332 \expandafter\collargs@r@i
3333 \fi
3334 }
3335 \def\collargs@r@verb#1#2{%
3336 \expandafter\collargs@r@i\detokenize{#1#2}%
3337 }
3338 \def\collargs@r@i#1#2#3.{%
```

We will need to use the `\collargs@empty` trick from type `u`, but with an additional twist: we need to insert it *after* the opening delimiter `#1`. To do this, we consume the opening delimiter by the “outer” collector below — we need to use the collector so that we get a nice error message when the opening delimiter is not present — and have this collector define the “inner” collector in the spirit of type `u`.

The outer collector has no parameters, it just requires the presence of the opening delimiter.

```
3339 \collargs@defcollector#1{%
```

The inner collector will grab everything up to the closing delimiter.

```
3340 \collargs@defusecollector####1#2{%
```

Append the collected argument `####1` to the list, wrapping it into the delimiters (`#1` and `#2`), but not before expanding its first token, which we know to be `\collargs@empty`.

```
3341 \collargs@wrap{#1#####1#2}%
3342 \expandafter\collargs@appendarg\expandafter{####1}%
3343 \collargs@#3.%
3344 }%
3345 \collargs@empty
3346 }%
```

Another complication: our delimited argument may be preceded by spaces. To replicate the argument tokens faithfully, we need to collect them before trying to grab the argument itself.

```
3347 \collargs@grabspaces\collargs@caller
3348 }
```

`\collargs@R` Discard the default and execute `r`.

```
3349 \def\collargs@R#1#2#3{\collargs@r#1#2}
```

`\collargs@d` Optional delimited argument. Very similar to `r`.

```
3350 \def\collargs@d{%
3351 \ifcollargs@verbatim
3352 \expandafter\collargs@d@verb
3353 \else
3354 \expandafter\collargs@d@i
3355 \fi
3356 }
3357 \def\collargs@d@verb#1#2{%
3358 \expandafter\collargs@d@i\detokenize{#1#2}%
3359 }
3360 \def\collargs@d@i#1#2#3.{%
```

This macro will be executed when the optional argument is not present. It simply closes the argument's group and reexecutes the central loop.

```
3361 \def\collargs@d@noopt{%
3362 \global\collargs@fix@requestedtrue
3363 \endgroup
3364 \collargs@#3.%
3365 }%
```

The collector(s) are exactly as for `r`.

```
3366 \collargs@defcollector#1{%
3367 \collargs@defusecollector####1#2{%
3368 \collargs@wrap{#1#####1#2}%
3369 \expandafter\collargs@appendarg\expandafter{####1}%
3370 \collargs@#3.%
3371 }%
3372 \collargs@empty
3373 }%
```

This macro will check, in conjunction with `\futurelet` below, whether the optional argument is present or not.

```

3374 \def\collargs@d@ii{%
3375   \ifx#1\collargs@temp
3376   \expandafter\collargs@caller
3377   \else
3378   \expandafter\collargs@d@noopt
3379   \fi
3380 }%

```

Whether spaces are allowed in front of this type of argument depends on the presence of modifier `!`.

```

3381 \collargs@maybegrabspaces{\futurelet\collargs@temp\collargs@d@ii}%
3382 }

```

`\collargs@D` Discard the default and execute `d`.

```

3383 \def\collargs@D#1#2#3{\collargs@d#1#2}

```

`\collargs@o` `o` is just `d` with delimiters `[` and `]`.

```

3384 \def\collargs@o{\collargs@d[]}

```

`\collargs@O` `O` is just `d` with delimiters `[` and `]` and the discarded default.

```

3385 \def\collargs@O#1{\collargs@d[]}

```

`\collargs@t` An optional token. Similar to `d`.

```

3386 \def\collargs@t{%
3387   \ifcollargs@verbatim
3388   \expandafter\collargs@t@verb
3389   \else
3390   \expandafter\collargs@t@i
3391   \fi
3392 }
3393 \def\collargs@t@space{ }
3394 \def\collargs@t@verb#1{%
3395   \let\collargs@t@space\collargs@otherspace
3396   \expandafter\collargs@t@i\expandafter{\detokenize{#1}}%
3397 }
3398 \def\collargs@t@i#1{%
3399   \expandafter\ifx\space#1%
3400   \expandafter\collargs@t@s
3401   \else
3402   \expandafter\collargs@t@I\expandafter#1%
3403   \fi
3404 }
3405 \def\collargs@t@s#1.{%
3406   \collargs@grabspaces{%
3407     \ifcollargs@grabspaces
3408     \collargs@appendarg{}%
3409     \else
3410     \expanded{%
3411       \noexpand\collargs@init@grabspaces
3412       \noexpand\collargs@appendarg{\collargs@grabbed@spaces}%
3413     }%
3414     \fi
3415     \collargs@#1.%
3416   }%
3417 }

```

```

3418 \def\collargs@t@I#1#2.{%
3419   \def\collargs@t@noopt{%
3420     \global\collargs@fix@requestedtrue
3421     \endgroup
3422     \collargs@#2.%
3423   }%
3424   \def\collargs@t@opt##1{%
3425     \collargs@appendarg{#1}%
3426     \collargs@#2.%
3427   }%
3428   \def\collargs@t@ii{%
3429     \ifx#1\collargs@temp
3430       \expandafter\collargs@t@opt
3431     \else
3432       \expandafter\collargs@t@noopt
3433     \fi
3434   }%
3435   \collargs@maybegrabspaces{\futurelet\collargs@temp\collargs@t@ii}%
3436 }
3437 \def\collargs@t@opt@space{%
3438   \expanded{\noexpand\collargs@t@opt{\space}\expandafter}\romannumeral-0%
3439 }%

```

`\collargs@s` The optional star is just a special case of `t`.

```

3440 \def\collargs@s{\collargs@t*}

```

`\collargs@m` Mandatory argument. Interestingly, here's where things get complicated, because we have to take care of several $\mathrm{T}_\mathrm{E}\mathrm{X}$ quirks.

```

3441 \def\collargs@m{%
3442   \ifcollargs@verbatim
3443     \expandafter\collargs@m@verb
3444   \else
3445     \expandafter\collargs@m@i
3446   \fi
3447 }

```

The non-verbatim mode. First, collect any spaces in front of the argument.

```

3448 \def\collargs@m@i#1.{%
3449   \collargs@grabspaces{\collargs@m@checkforgroup#1.}%
3450 }

```

Is the argument in braces or not?

```

3451 \def\collargs@m@checkforgroup#1.{%
3452   \edef\collargs@action{\unexpanded{\collargs@m@checkforgroup@i#1.}}%
3453   \futurelet\collargs@token\collargs@action
3454 }
3455 \def\collargs@m@checkforgroup@i{%
3456   \ifcat\noexpand\collargs@token\bgroup
3457     \expandafter\collargs@m@group
3458   \else
3459     \expandafter\collargs@m@token
3460   \fi
3461 }

```

The argument is given in braces, so we put them back around it (`\collargs@wrap`) when appending to the storage.

```

3462 \def\collargs@m@group#1.{%
3463   \collargs@defusecollector##1{%

```

```

3464 \collargs@wrap{####1}%
3465 \collargs@appendarg{##1}%
3466 \collargs@#1.%
3467 }%
3468 }

```

The argument is a single token, we append it to the storage as is.

```

3469 \def\collargs@m@token#1.{%
3470 \collargs@defusecollector##1{%
3471 \collargs@appendarg{##1}%
3472 \collargs@#1.%
3473 }%
3474 }

```

The verbatim mode. Again, we first collect any spaces in front of the argument.

```

3475 \def\collargs@m@verb#1.{%
3476 \collargs@grabspaces{\collargs@m@verb@checkforgroup#1.}%
3477 }

```

We want to check whether we're dealing with a braced argument. We're in the verbatim mode, but are braces verbatim as well? In other words, are we in `verbatim` or `verb` mode? In the latter case, braces are regular, so we redirect to the regular mode.

```

3478 \def\collargs@m@verb@checkforgroup{%
3479 \ifcollargs@verbatimbraces
3480 \expandafter\collargs@m@verb@checkforgroup@i
3481 \else
3482 \expandafter\collargs@m@checkforgroup
3483 \fi
3484 }

```

Is the argument in verbatim braces?

```

3485 \def\collargs@m@verb@checkforgroup@i#1.{%
3486 \def\collargs@m@verb@checkforgroup@ii{\collargs@m@verb@checkforgroup@iii#1.}%
3487 \futurelet\collargs@temp\collargs@m@verb@checkforgroup@ii
3488 }
3489 \def\collargs@m@verb@checkforgroup@iii#1.{%
3490 \expandafter\ifx\collargs@other@bgroup\collargs@temp

```

Yes, the argument is in (verbatim) braces.

```

3491 \expandafter\collargs@m@verb@group
3492 \else

```

We need to manually check whether the following token is a (verbatim) closing brace, and throw an error if it is.

```

3493 \expandafter\ifx\collargs@other@egroup\collargs@temp
3494 \expandafter\expandafter\expandafter\collargs@m@verb@egrouperror
3495 \else

```

The argument is a single token.

```

3496 \expandafter\expandafter\expandafter\collargs@m@v@token
3497 \fi
3498 \fi
3499 #1.%
3500 }
3501 \def\collargs@m@verb@egrouperror#1.{%
3502 \PackageError{collargs}{%
3503 Argument of \expandafter\string\collargs@caller\space has an extra
3504 \iffalse{\else\string}}}%
3505 }

```

A single-token verbatim argument.

```
3506 \def\collargs@m@v@token#1.#2{%
```

Is it a control sequence? (Macro `\collargs@cc` recalls the “outside” category code of character `#1`; see section 8.2.5.)

```
3507 \ifnum\collargs@cc{#2}=0
3508   \expandafter\collargs@m@v@token@cs
3509 \else
3510   \expandafter\collargs@m@token
3511 \fi
3512 #1.#2%
3513 }
```

Is it a one-character control sequence?

```
3514 \def\collargs@m@v@token@cs#1.#2#3{%
3515   \ifnum\collargs@cc{#3}=11
3516     \expandafter\collargs@m@v@token@cs@letter
3517   \else
3518     \expandafter\collargs@m@v@token@cs@nonletter
3519   \fi
3520 #1.#2#3%
3521 }
```

Store `\<token>`.

```
3522 \def\collargs@m@v@token@cs@nonletter#1.#2#3{%
3523   \collargs@appendarg{#2#3}%
3524   \collargs@#1.%
3525 }
```

Store `\` to a temporary register, we’ll parse the control sequence name now.

```
3526 \def\collargs@m@v@token@cs@letter#1.#2{%
3527   \collargsArg{#2}%
3528   \def\collargs@tempa{#1}%
3529   \collargs@m@v@token@cs@letter@i
3530 }
```

Append a letter to the control sequence.

```
3531 \def\collargs@m@v@token@cs@letter@i#1{%
3532   \ifnum\collargs@cc{#1}=11
3533     \toksapp\collargsArg{#1}%
3534     \expandafter\collargs@m@v@token@cs@letter@i
3535   \else
```

Finish, returning the non-letter to the input stream.

```
3536     \expandafter\collargs@m@v@token@cs@letter@ii\expandafter#1%
3537   \fi
3538 }
```

Store the verbatim control sequence.

```
3539 \def\collargs@m@v@token@cs@letter@ii{%
3540   \expanded{%
3541     \unexpanded{%
3542       \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
3543     }%
3544     \noexpand\collargs@\expandonce\collargs@tempa.%
3545   }%
3546 }
```

The verbatim mandatory argument is delimited by verbatim braces. We have to use the heavy machinery adapted from `cprotect`.

```

3547 \def\collargs@m@verb@group#1.#2{%
3548   \let\collargs@begintag\collargs@other@bgroup
3549   \let\collargs@endtag\collargs@other@egroup
3550   \def\collargs@tagarg{%
3551     \def\collargs@commandatend{\collargs@m@verb@group@i#1.}%
3552     \collargs@readContent
3553   }

```

This macro appends the result given by the heavy machinery, waiting for us in macro `\collargsArg`, to `\collargsArgs`, but not before dressing it up (via `\collargs@wrap`) in a pair of verbatim braces.

```

3554 \def\collargs@m@verb@group@i{%
3555   \edef\collargs@temp{%
3556     \collargs@other@bgroup\unexpanded{##1}\collargs@other@egroup}%
3557   \expandafter\collargs@wrap\expandafter{\collargs@temp}%
3558   \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
3559   \collargs@
3560 }

```

`\collargs@g` An optional group: same as `m`, but we simply bail out if we don't find the group character.

```

3561 \def\collargs@g{%
3562   \def\collargs@m@token{%
3563     \global\collargs@fix@requestedtrue
3564     \endgroup
3565     \collargs@
3566   }%
3567   \let\collargs@m@v@token\collargs@m@token
3568   \collargs@m
3569 }

```

`\collargs@G` Discard the default and execute `g`.

```

3570 \def\collargs@G#1{\collargs@g}

```

`\collargs@v` Verbatim argument. The code is executed in the group, deploying `\collargsVerbatim`. The grouping characters are always set to braces, to mimick `xparse` perfectly.

```

3571 \def\collargs@v#1.{%
3572   \begingroup
3573   \collargsBraces{}}%
3574   \collargsVerbatim
3575   \collargs@grabspace{\collargs@v@i#1.}%
3576 }
3577 \def\collargs@v@i#1.#2{%
3578   \expandafter\ifx\collargs@other@bgroup#2%

```

If the first token we see is an opening brace, use the `cprotect` adaptation to grab the group.

```

3579   \let\collargs@begintag\collargs@other@bgroup
3580   \let\collargs@endtag\collargs@other@egroup
3581   \def\collargs@tagarg{%
3582     \def\collargs@commandatend{%
3583       \edef\collargs@temp{%
3584         \collargs@other@bgroup\unexpanded{####1}\collargs@other@egroup}%
3585       \expandafter\collargs@wrap\expandafter{\collargs@temp}%
3586       \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
3587       \endgroup
3588       \collargs@#1.%

```



```

3589 }%
3590 \expandafter\collargs@readContent
3591 \else

```

Otherwise, the verbatim argument is delimited by two identical characters (#2).

```

3592 \collargs@defcollector##1#2{%
3593 \collargs@wrap{#2###1#2}%
3594 \collargs@appendarg{##1}%
3595 \endgroup
3596 \collargs@#1.%
3597 }%
3598 \expandafter\collargs@caller
3599 \fi
3600 }

```

`\collargs@b` Environments. Here's where all hell breaks loose. We survive by adapting some code from Bruno Le Floch's `cprotect`. We first define the environment-related keys, then provide the handler code, and finish with the adaptation of `cprotect`'s environment-grabbing code.

The argument type `b` token may be followed by a braced environment name (in the argument specification).

```

3601 \def\collargs@b{%
3602 \collargs@ifnextcat\bgroup\collargs@bg\collargs@bi
3603 }
3604 \def\collargs@bg#1{%
3605 \edef\collargs@b@envname{#1}%
3606 \collargs@bi
3607 }
3608 \def\collargs@bi#1.{%

```

Convert the environment name to verbatim if necessary.

```

3609 \ifcollargs@verbatim
3610 \edef\collargs@b@envname{\detokenize\expandafter{\collargs@b@envname}}%
3611 \fi

```

This is a format-specific macro which sets up `\collargs@begintag` and `\collargs@endtag`.

```

3612 \collargs@bi@defCPTbeginend
3613 \edef\collargs@tagarg{%
3614 \ifcollargs@verbatimbraces
3615 \else
3616 \ifcollargsIgnoreOtherTags
3617 \collargs@b@envname
3618 \fi
3619 \fi
3620 }%

```

Run this after collecting the body.

```

3621 \def\collargs@commandatend{%

```

In \LaTeX , we might, depending on the verbatim mode, need to check whether the environment name is correct.

```

3622 <latex> \collargs@bii

```

In plain \TeX and \ConTeXt , we can skip directly to `\collargs@biii`.

```

3623 <plain,context> \collargs@biii
3624 #1.%
3625 }%

```

Collect the environment body, but first, put any grabbed spaces back into the input stream.

```
3626 \collargs@reinsert@spaces\collargs@readContent
3627 }
3628 <*/latex>
```

In L^AT_EX in the regular and the partial verbatim mode, we search for `\begin/\end` — as we cannot search for braces — either as control sequences in the regular mode, or as strings in the partial verbatim mode. (After search, we will have to check whether the argument of `\begin/\end` matches our environment name.) In the full verbatim mode, we can search for the entire string `\begin/\end{<name>}`.

```
3629 \def\collargs@bi@defCPTbeginend{%
3630   \edef\collargs@begintag{%
3631     \ifcollargs@verbatim
3632       \expandafter\string
3633     \else
3634       \expandafter\noexpand
3635     \fi
3636     \begin
3637     \ifcollargs@verbatimbraces
3638       \collargs@other@bgroup\collargs@b@envname\collargs@other@egroup
3639     \fi
3640   }%
3641   \edef\collargs@endtag{%
3642     \ifcollargs@verbatim
3643       \expandafter\string
3644     \else
3645       \expandafter\noexpand
3646     \fi
3647     \end
3648     \ifcollargs@verbatimbraces
3649       \collargs@other@bgroup\collargs@b@envname\collargs@other@egroup
3650     \fi
3651   }%
3652 }
3653 </latex>
3654 <*plain, context>
```

We can search for the entire `\<name>/\end<name>` (in T_EX) or `\start<name>/\stop<name>` (in ConT_EXt), either as a control sequence (in the regular mode), or as a string (in the verbatim modes).

```
3655 \def\collargs@bi@defCPTbeginend{%
3656   \edef\collargs@begintag{%
3657     \ifcollargs@verbatim
3658       \expandafter\expandafter\expandafter\string
3659     \else
3660       \expandafter\expandafter\expandafter\noexpand
3661     \fi
3662     \csname
3663 <context>      start%
3664       \collargs@b@envname
3665     \endcsname
3666   }%
3667   \edef\collargs@endtag{%
3668     \ifcollargs@verbatim
3669       \expandafter\expandafter\expandafter\string
3670     \else
3671       \expandafter\expandafter\expandafter\noexpand
3672     \fi
3673     \csname
3674 <plain>      end%
```

```

3675 <context>      stop%
3676               \collargs@b@envname
3677               \endcsname
3678             }%
3679 }
3680 </plain, context>
3681 <*latex>

```

Check whether we're in front of the (braced) environment name (in L^AT_EX), and consume it.

```

3682 \def\collargs@bii{%
3683   \ifcollargs@verbatimbraces
3684     \expandafter\collargs@biii
3685   \else
3686     \ifcollargsIgnoreOtherTags

```

We shouldn't check the name in this case, because it was already checked, and consumed.

```

3687     \expandafter\expandafter\expandafter\collargs@biii
3688   \else
3689     \expandafter\expandafter\expandafter\collargs@b@checkend
3690   \fi
3691 \fi
3692 }
3693 \def\collargs@b@checkend#1.{%
3694   \collargs@grabspaces{\collargs@b@checkend@i#1.}%
3695 }
3696 \def\collargs@b@checkend@i#1.#2{%
3697   \def\collargs@temp{#2}%
3698   \ifx\collargs@temp\collargs@b@envname
3699   \else
3700     \collargs@b@checkend@error
3701   \fi
3702   \collargs@biii#1.%
3703 }
3704 \def\collargs@b@checkend@error{%
3705   \PackageError{collargs}{Environment "\collargs@b@envname" ended as
3706     "\collargs@temp"}{ }%
3707 }
3708 </latex>

```

This macro stores the collected body.

```

3709 \def\collargs@biii{%

```

Define the wrapper macro (`\collargs@temp`).

```

3710   \collargs@b@def@wrapper

```

Execute `\collargs@appendarg` to append the body to the list. Expand the wrapper in `\collargs@temp` first and the body in `\collargsArg` next.

```

3711   \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%

```

Reexecute the central loop.

```

3712   \collargs@
3713 }
3714 \def\collargs@b@def@wrapper{%
3715 <latex>   \edef\collargs@temp{\collargs@b@envname}}%
3716   \edef\collargs@temp{%

```

Was the begin-tag requested?

```

3717   \ifcollargsBeginTag

```

`\collargs@begintag` is already adapted to the format and the verbatim mode.

```
3718 \expandonce\collargs@begintag
```

Add the braced environment name in \LaTeX in the regular and partial verbatim mode.

```
3719 *latex>
3720 \ifcollargs@verbatimbraces\else\collargs@temp\fi
3721 /latex>
3722 \fi
```

This is the body.

```
3723 #####1%
```

Rinse and repeat for the end-tag.

```
3724 \ifcollargsEndTag
3725 \expandonce\collargs@endtag
3726 *latex>
3727 \ifcollargs@verbatimbraces\else\collargs@temp\fi
3728 /latex>
3729 \fi
3730 }%
3731 \expandafter\collargs@wrap\expandafter{\collargs@temp}%
3732 }
```

`\collargs@readContent` This macro, which is an adaptation of `cprotect`'s environment-grabbing code, collects some delimited text, leaving the result in `\collargsArg`. Before calling it, one must define the following macros: `\collargs@begintag` and `\collargs@endtag` are the content delimiters; `\collargs@tagarg`, if non-empty, is the token or grouped text which must follow a delimiter to be taken into account; `\collargs@commandatend` is the command that will be executed once the content is collected.

```
3733 \def\collargs@readContent{%
```

Define macro which will search for the first begin-tag.

```
3734 \ifcollargs@long\long\fi
3735 \collargs@CPT@def\collargs@gobbleOneB\collargs@begintag{%
```

Assign the collected tokens into a register. The first token in `##1` will be `\collargs@empty`, so we expand to get rid of it.

```
3736 \toks0\expandafter{##1}%
```

`cprotect` simply grabs the token following the `\collargs@begintag` with a parameter. We can't do this, because we need the code to work in the non-verbatim mode, as well, and we might stumble upon a brace there. So we take a peek.

```
3737 \futurelet\collargs@temp\collargs@gobbleOneB@i
3738 }%
```

Define macro which will search for the first end-tag. We make it long if so required (by +).

```
3739 \ifcollargs@long\long\fi
3740 \collargs@CPT@def\collargs@gobbleUntilE\collargs@endtag{%
```

Expand `\collargs@empty` at the start of `##1`.

```
3741 \expandafter\toksapp\expandafter0\expandafter{##1}%
3742 \collargs@gobbleUntilE@i
3743 }%
```

Initialize.

```
3744 \collargs@begins=0\relax
3745 \collargsArg{}%
3746 \toks0{}%
```

We will call `\collargs@gobbleUntilE` via the caller control sequence.

```
3747 \collargs@letusecollector\collargs@gobbleUntilE
```

We insert `\collargs@empty` to avoid the potential debracing problem.

```
3748 \collargs@empty
3749 }
```

How many begin-tags do we have opened?

```
3750 \newcount\collargs@begins
```

An auxiliary macro which `\defs #1` so that it will grab everything up until `#2`. Additional parameters may be present before the definition.

```
3751 \def\collargs@CPT@def#1#2{%
3752 \expandafter\def\expandafter#1%
3753 \expandafter##\expandafter1#2%
3754 }
```

A quark guard.

```
3755 \def\collargs@qend{\collargs@qend}
```

This macro will collect the “environment”, leaving the result in `\collargsArg`. It expects `\collargs@begintag`, `\collargs@endtag` and `\collargs@commandatend` to be set.

```
3756 \def\collargs@gobbleOneB@i{%
3757 \def\collargs@begins@increment{1}%
3758 \ifx\collargs@qend\collargs@temp
```

We have reached the fake begin-tag. Note that we found the end-tag.

```
3759 \def\collargs@begins@increment{-1}%
```

Gobble the quark guard.

```
3760 \expandafter\collargs@gobbleOneB@v
3761 \else
```

Append the real begin-tag to the temporary tokens.

```
3762 \etoksapp0{\expandonce\collargs@begintag}%
3763 \expandafter\collargs@gobbleOneB@ii
3764 \fi
3765 }%
```

Do we have to check the tag argument (i.e. the environment name after `\begin`)?

```
3766 \def\collargs@gobbleOneB@ii{%
3767 \expandafter\ifx\expandafter\relax\collargs@tagarg\relax
3768 \expandafter\collargs@gobbleOneB@vi
3769 \else
```

Yup, so let's (carefully) collect the tag argument.

```
3770 \expandafter\collargs@gobbleOneB@iii
3771 \fi
3772 }
3773 \def\collargs@gobbleOneB@iii{%
3774 \collargs@grabspaces{%
3775 \collargs@letusecollector\collargs@gobbleOneB@iv
3776 }%
3777 }
3778 \def\collargs@gobbleOneB@iv#1{%
3779 \def\collargs@temp{#1}%
3780 \ifx\collargs@temp\collargs@tagarg
```

This is the tag argument we've been waiting for!

```
3781 \else
```

Nope, this `\begin` belongs to someone else.

```
3782 \def\collargs@begins@increment{0}%
3783 \fi
```

Whatever the result was, we have to append the gobbled group to the temporary toks.

```
3784 \etoksapp0{\collargs@grabbed@spaces\unexpanded{{#1}}}%
3785 \collargs@init@grabspaces
3786 \collargs@gobbleOneB@vi
3787 }
3788 \def\collargs@gobbleOneB@v#1{\collargs@gobbleOneB@vi}
3789 \def\collargs@gobbleOneB@vi{%
```

Store.

```
3790 \etoksapp\collargsArg{\the\toks0}%
```

Advance the begin-tag counter.

```
3791 \advance\collargs@begins\collargs@begins@increment\relax
```

Find more begin-tags, unless this was the final one.

```
3792 \ifnum\collargs@begins@increment=-1
3793 \else
3794 \expandafter\collargs@gobbleOneB\expandafter\collargs@empty
3795 \fi
3796 }
3797 \def\collargs@gobbleUntilE@i{%
```

Do we have to check the tag argument (i.e. the environment name after `\end`)?

```
3798 \expandafter\ifx\expandafter\relax\collargs@tagarg\relax
3799 \expandafter\collargs@gobbleUntilE@iv
3800 \else
```

Yup, so let's (carefully) collect the tag argument.

```
3801 \expandafter\collargs@gobbleUntilE@ii
3802 \fi
3803 }
3804 \def\collargs@gobbleUntilE@ii{%
3805 \collargs@grabspaces{%
3806 \collargs@letusecollector\collargs@gobbleUntilE@iii
3807 }%
3808 }
```

```

3809 \def\collargs@gobbleUntilE@iii#1{%
3810 \etoksapp0{\collargs@grabbed@spaces}%
3811 \collargs@init@grabspaces
3812 \def\collargs@tempa{#1}%
3813 \ifx\collargs@tempa\collargs@tagarg

```

This is the tag argument we've been waiting for!

```

3814 \expandafter\collargs@gobbleUntilE@iv
3815 \else

```

Nope, this `\end` belongs to someone else. Insert the end tag plus the tag argument, and collect until the next `\end`.

```

3816 \expandafter\toksapp\expandafter0\expandafter{\collargs@endtag{#1}}%
3817 \expandafter\collargs@letusecollector\expandafter\collargs@gobbleUntilE
3818 \fi
3819 }
3820 \def\collargs@gobbleUntilE@iv{%

```

Invoke `\collargs@gobbleOneB` with the collected material, plus a fake begin-tag and a quark guard.

```

3821 \ifcollargsIgnoreNesting
3822 \expandafter\collargsArg\expandafter{\the\toks0}%
3823 \expandafter\collargs@commandatend
3824 \else
3825 \expandafter\collargs@gobbleUntilE@v
3826 \fi
3827 }
3828 \def\collargs@gobbleUntilE@v{%
3829 \expanded{%
3830 \noexpand\collargs@letusecollector\noexpand\collargs@gobbleOneB
3831 \noexpand\collargs@empty
3832 \the\toks0

```

Add a fake begin-tag and a quark guard.

```

3833 \expandonce\collargs@begintag
3834 \noexpand\collargs@qend
3835 }%
3836 \ifnum\collargs@begins<0
3837 \expandafter\collargs@commandatend
3838 \else
3839 \etoksapp\collargsArg{%
3840 \expandonce\collargs@endtag
3841 \expandafter\ifx\expandafter\relax\collargs@tagarg\relax\else{%
3842 \expandonce\collargs@tagarg}\fi
3843 }%
3844 \toks0={}%
3845 \expandafter\collargs@letusecollector\expandafter\collargs@gobbleUntilE
3846 \expandafter\collargs@empty
3847 \fi
3848 }

```

`\collargs@e` Embellishments. Each embellishment counts as an argument, in the sense that we will execute `\collargs@appendarg`, with all the processors, for each embellishment separately.

```

3849 \def\collargs@e{%

```

We open an extra group, because `\collargs@appendarg` will close a group for each embellishment.

```

3850 \global\collargs@fix@requestedtrue
3851 \begingroup

```



```

3852 \ifcollargs@verbatim
3853   \expandafter\collargs@e@verbatim
3854 \else
3855   \expandafter\collargs@e@i
3856 \fi
3857 }

```

Detokenize the embellishment tokens in the verbatim mode.

```

3858 \def\collargs@e@verbatim#1{%
3859   \expandafter\collargs@e@i\expandafter{\detokenize{#1}}%
3860 }

```

Ungroup the embellishment tokens, separating them from the rest of the argument specification by a dot.

```

3861 \def\collargs@e@i#1{\collargs@e@ii#1.}

```

We now have embellishment tokens in #1 and the rest of the argument specification in #2. Let's grab spaces first.

```

3862 \def\collargs@e@ii#1.#2.{%
3863   \collargs@grabspaces{\collargs@e@iii#1.#2.}%
3864 }

```

What's the argument token?

```

3865 \def\collargs@e@iii#1.#2.{%
3866   \def\collargs@e@iv{\collargs@e@v#1.#2.}%
3867   \futurelet\collargs@temp\collargs@e@iv
3868 }

```

If it is a open or close group character, we surely don't have an embellishment.

```

3869 \def\collargs@e@v{%
3870   \ifcat\noexpand\collargs@temp\bgroup\relax
3871     \let\collargs@marshal\collargs@e@z
3872   \else
3873     \ifcat\noexpand\collargs@temp\egroup\relax
3874       \let\collargs@marshal\collargs@e@z
3875     \else
3876       \let\collargs@marshal\collargs@e@vi
3877     \fi
3878   \fi
3879   \collargs@marshal
3880 }

```

We borrow the “Does #1 occur within #2?” macro from `pgfutil-common`, but we fix it by executing `\collargs@in@@` in a braced group. This will prevent an `&` in an argument to function as an alignment character; the minor price to pay is that we assign the conditional globally.

```

3881 \newif\ifcollargs@in@
3882 \def\collargs@in@#1#2{%
3883   \def\collargs@in@@##1##2##3\collargs@in@@{%
3884     \ifx\collargs@in@@##2\global\collargs@in@false\else\global\collargs@in@true\fi
3885   }%
3886   {\collargs@in@@##2#1\collargs@in@\collargs@in@@}%
3887 }

```

Let's see whether the following token, now #3, is an embellishment token.

```

3888 \def\collargs@e@vi#1.#2.#3{%
3889   \collargs@in@{#3}{#1}%
3890   \ifcollargs@in@

```

```

3891 \expandafter\collargs@e@vii
3892 \else
3893 \expandafter\collargs@e@z
3894 \fi
3895 #1.#2.#3%
3896 }

```

#3 is the current embellishment token. We'll collect its argument using `\collargs@m`, but to do that, we have to (locally) redefine `\collargs@appendarg` and `\collargs@`, which get called by `\collargs@m`.

```

3897 \def\collargs@e@vii#1.#2.#3{%

```

We'll have to execute the original `\collargs@appendarg` later, so let's remember it. The temporary `\collargs@appendarg` simply stores the collected argument into `\collargsArg` — we'll do the processing etc. later.

```

3898 \let\collargs@real@appendarg\collargs@appendarg
3899 \def\collargs@appendarg##1{\collargsArg{##1}}%

```

Once `\collargs@m` is done, it will call the redefined `\collargs@` and thereby get us back into this handler.

```

3900 \def\collargs@{\collargs@e@viii#1.#3}%
3901 \collargs@m#2.%
3902 }

```

The parameters here are as follows. #1 are the embellishment tokens, and #2 is the current embellishment token; these get here via our local redefinition of `\collargs@` in `\collargs@e@vii`. #3 are the rest of the argument specification, which is put behind control sequence `\collargs@` by the `m` handler.

```

3903 \def\collargs@e@viii#1.#2#3.{%

```

Our wrapper puts the current embellishment token in front of the collected embellishment argument. Note that if the embellishment argument was in braces, `\collargs@m` has already set one wrapper (which will apply first).

```

3904 \collargs@wrap{#2##1}%

```

We need to get rid of the current embellishment from embellishments, not to catch the same embellishment twice.

```

3905 \def\collargs@e@ix##1#2{\collargs@e@x##1}%
3906 \collargs@e@ix#1.#3.%
3907 }

```

When this is executed, the input stream starts with the (remaining) embellishment tokens, followed by a dot, then the rest of the argument specification, also followed by a dot.

```

3908 \def\collargs@e@x{%

```

Process the argument and append it to the storage.

```

3909 \expandafter\collargs@real@appendarg\expandafter{\the\collargsArg}%

```

`\collargs@real@appendarg` has closed a group, so we open it again, and start looking for another embellishment token in the input stream.

```

3910 \begingroup
3911 \collargs@e@ii
3912 }

```

The first argument token is not an embellishment token. We finish by consuming the list of embellishment tokens, closing the two groups opened by this handler, and reexecuting the central loop.

```
3913 \def\collargs@e#1.{\endgroup\endgroup\collargs@}
```

`\collargs@E` Discard the defaults and execute e.

```
3914 \def\collargs@E#1#2{\collargs@e{#1}}
```

8.2.5 The verbatim modes

`\collargsVerbatim` These macros set the two verbatim-related conditionals, `\ifcollargs@verbatim` and `\collargsVerb` `\ifcollargs@verbatimbraces`, and then call `\collargs@make@verbatim` to effect the requested category code changes (among other things). A group should be opened prior to executing either of them. After execution, they are redefined to minimize the effort needed to enter into another mode in an embedded group. Below, we first define all the possible transitions.

```
3915 \let\collargs@NoVerbatimAfterNoVerbatim\relax
3916 \def\collargs@VerbAfterNoVerbatim{%
3917   \collargs@verbatimtrue
3918   \collargs@verbatimbracesfalse
3919   \collargs@make@verbatim
3920   \collargs@after{Verb}%
3921 }
3922 \def\collargs@VerbatimAfterNoVerbatim{%
3923   \collargs@verbatimtrue
3924   \collargs@verbatimbracestrue
3925   \collargs@make@verbatim
3926   \collargs@after{Verbatim}%
3927 }
3928 \def\collargs@NoVerbatimAfterVerb{%
3929   \collargs@verbatimfalse
3930   \collargs@verbatimbracesfalse
3931   \collargs@make@other@groups
3932   \collargs@make@no@verbatim
3933   \collargs@after{NoVerbatim}%
3934 }
3935 \def\collargs@VerbAfterVerb{%
3936   \collargs@make@other@groups
3937 }
3938 \def\collargs@VerbatimAfterVerb{%
3939   \collargs@verbatimbracestrue
3940   \collargs@make@other@groups
```

Process the lists of grouping characters, created by `\collargs@make@verbatim`, making these characters of category “other”.

```
3941 \def\collargs@do##1{\catcode##1=12 }%
3942 \collargs@bgroups
3943 \collargs@egroups
3944 \collargs@after{Verbatim}%
3945 }%
3946 \let\collargs@NoVerbatimAfterVerbatim\collargs@NoVerbatimAfterVerb
3947 \def\collargs@VerbAfterVerbatim{%
3948   \collargs@verbatimbracesfalse
3949   \collargs@make@other@groups
```

Process the lists of grouping characters, created by `\collargs@make@verbatim`, making these characters be of their normal category.

```
3950 \def\collargs@do##1{\catcode##1=1 }%
3951 \collargs@bgroups
```

```

3952 \def\collargs@do##1{\catcode##1=2 }%
3953 \collargs@egroups
3954 \collargs@after{Verb}%
3955 }%
3956 \let\collargs@VerbatimAfterVerbatim\collargs@VerbAfterVerb

```

This macro expects #1 to be the mode just entered (Verbatim, Verb or NoVerbatim), and points macros \collargsVerbatim, \collargsVerb and \collargsNoVerbatim to the appropriate transition macro.

```

3957 \def\collargs@after#1{%
3958 \letcs\collargsVerbatim{collargs@VerbatimAfter#1}%
3959 \letcs\collargsVerb{collargs@VerbAfter#1}%
3960 \letcs\collargsNoVerbatim{collargs@NoVerbatimAfter#1}%
3961 }

```

The first transition is always from the non-verbatim mode.

```

3962 \collargs@after{NoVerbatim}

```

\collargs@bgroups Initialize the lists of the current grouping characters used in the redefinitions of macros **\collargs@egroups** \collargsVerbatim and \collargsVerb above. Each entry is of form \collargs@do{*character code*}. These lists will be populated by \collargs@make@verbatim. They may be local, as they only used within the group opened for a verbatim environment.

```

3963 \def\collargs@bgroups{%
3964 \def\collargs@egroups{%

```

\collargs@cc This macro recalls the category code of character #1. In LuaTeX, we simply look up the category code in the original category code table; in other engines, we have stored the original category code into \collargs@cc@*character code* by \collargs@make@verbatim. (Note that #1 is a character, not a number.)

```

3965 \ifdefined\luatexversion
3966 \def\collargs@cc#1{%
3967 \directlua{tex.sprint(tex.getcatcode(\collargs@catcodetable@original,
3968 \the\numexpr\expandafter`\csname#1\endcsname\relax))}%
3969 }
3970 \else
3971 \def\collargs@cc#1{%
3972 \ifcsname collargs@cc@\the\numexpr\expandafter`\csname#1\endcsname\endcsname
3973 \csname collargs@cc@\the\numexpr\expandafter`\csname#1\endcsname\endcsname
3974 \else
3975 12%
3976 \fi
3977 }
3978 \fi

```

\collargs@other@bgroup Macros \collargs@other@bgroup and \collargs@other@egroup hold the characters **\collargs@other@egroup** of category code “other” which will play the role of grouping characters in the **\collargsBraces** full verbatim mode. They are usually defined when entering a verbatim mode in \collargs@make@verbatim, but may be also set by the user via \collargsBraces (it is not even necessary to select characters which indeed have the grouping function in the outside category code regime). The setting process is indirect: executing \collargsBraces merely sets \collargs@make@other@groups, which gets executed by the subsequent \collargsVerbatim, \collargsVerb or \collargsNoVerbatim (either directly or via \collargs@make@verbatim).

```

3979 \def\collargsBraces#1{%
3980 \expandafter\collargs@braces@i\detokenize{#1}\relax
3981 }
3982 \def\collargs@braces@i#1#2#3\relax{%

```

```

3983 \def\collargs@make@other@groups{%
3984   \def\collargs@other@bgroup{#1}%
3985   \def\collargs@other@egroup{#2}%
3986 }%
3987 }
3988 \def\collargs@make@other@groups{}

```

`\collargs@catcodetable@verbatim` We declare several new catcode tables in LuaTeX, the most important `\catcodetable@atletter` one being `\collargs@catcodetable@verbatim`, where all characters have `\collargs@catcodetable@initex` category code 12. We only need the other two tables in some formats: `\collargs@catcodetable@atletter` holds the catcode in effect at the time of loading the package, and `\collargs@catcodetable@initex` is the iniTeX table.

```

3989 \ifdefined\luatexversion
3990 (*latex, context)
3991 \newcatcodetable\collargs@catcodetable@verbatim
3992 <latex> \let\collargs@catcodetable@atletter\catcodetable@atletter
3993 <context> \newcatcodetable\collargs@catcodetable@atletter
3994 </latex, context>
3995 (*plain)
3996 \ifdefined\collargs@catcodetable@verbatim\else
3997   \chardef\collargs@catcodetable@verbatim=4242
3998 \fi
3999 \chardef\collargs@catcodetable@atletter=%
4000   \number\numexpr\collargs@catcodetable@verbatim+1\relax
4001 \chardef\collargs@catcodetable@initex=%
4002   \number\numexpr\collargs@catcodetable@verbatim+2\relax
4003 \initcatcodetable\collargs@catcodetable@initex
4004 </plain>
4005 <plain, context> \savecatcodetable\collargs@catcodetable@atletter
4006 \begingroup
4007 \@firstofone{%
4008 <latex> \catcodetable\catcodetable@initex
4009 <plain> \catcodetable\collargs@catcodetable@initex
4010 <context> \catcodetable\inicatcodes
4011 \catcode`\=12
4012 \catcode13=12
4013 \catcode0=12
4014 \catcode32=12
4015 \catcode`\%=12
4016 \catcode127=12
4017 \def\collargs@do#1{\catcode#1=12 }%
4018 \collargs@forrange{`a}{`z}%
4019 \collargs@forrange{`A}{`Z}%
4020 \savecatcodetable\collargs@catcodetable@verbatim
4021 \endgroup
4022 }%
4023 \fi

```

`verbatim ranges` This key and macro set the character ranges to which the verbatim mode will apply (in `\collargsVerbatimRanges` pdfTeX and XeTeX), or which will be inspected for grouping and comment characters `\collargs@verbatim@ranges` (in LuaTeX). In pdfTeX, the default value 0–255 should really remain unchanged.

```

4024 \collargsSet{
4025   verbatim ranges/.store in=\collargs@verbatim@ranges,
4026 }
4027 \def\collargsVerbatimRanges#1{\def\collargs@verbatim@ranges{#1}}
4028 \def\collargs@verbatim@ranges{0-255}

```

`\collargs@make@verbatim` This macro changes the category code of all characters to “other” — except the grouping characters in the partial verbatim mode. While doing that, it also stores (unless we’re in LuaTeX) the current category codes into `\collargs@cc@<character code>` (easily recallable by

`\collargs@cc`), redefines the “primary” grouping characters `\collargs@make@other@bgroup` and `\collargs@make@other@egroup` if necessary, and “remembers” the grouping characters (storing them into `\collargs@bgroups` and `\collargs@egroups`) and the comment characters (storing them into `\collargs@comments`).

In LuaTeX, we can use catcode tables, so we change the category codes by switching to category code table `\collargs@catcodetable@verbatim`. In other engines, we have to change the codes manually. In order to offer some flexibility in XeTeX, we perform the change for characters in `verbatim` ranges.

```

4029 \ifdefined\luatexversion
4030   \def\collargs@make@verbatim{%
4031     \directlua{%
4032       for from, to in string.gmatch(
4033         "\luaescapestring{\collargs@verbatim@ranges}",
4034         "(\collargs@percentchar d+)-(\collargs@percentchar d+)"
4035       ) do
4036         for char = tex.round(from), tex.round(to) do
4037           catcode = tex.catcode[char]

```

For category codes 1, 2 and 14, we have to call macros `\collargs@make@verbatim@bgroup`, `\collargs@make@verbatim@egroup` and `\collargs@make@verbatim@comment`, same as for engines other than LuaTeX.

```

4038       if catcode == 1 then
4039         tex.sprint(
4040           \number\collargs@catcodetable@atletter,
4041           "\noexpand\collargs@make@verbatim@bgroup{" .. char .. "}")
4042       elseif catcode == 2 then
4043         tex.sprint(
4044           \number\collargs@catcodetable@atletter,
4045           "\noexpand\collargs@make@verbatim@egroup{" .. char .. "}")
4046       elseif catcode == 14 then
4047         tex.sprint(
4048           \number\collargs@catcodetable@atletter,
4049           "\noexpand\collargs@make@verbatim@comment{" .. char .. "}")
4050       end
4051     end
4052   end
4053 }%
4054 \edef\collargs@catcodetable@original{\the\catcodetable}%
4055 \catcodetable\collargs@catcodetable@verbatim

```

Even in LuaTeX, we switch between the `verbatim` braces regimes by hand.

```

4056   \ifcollargs@verbatimbraces
4057   \else
4058     \def\collargs@do##1{\catcode##1=1\relax}%
4059     \collargs@bgroups
4060     \def\collargs@do##1{\catcode##1=2\relax}%
4061     \collargs@egroups
4062   \fi
4063 }
4064 \else

```

The non-LuaTeX version:

```

4065   \def\collargs@make@verbatim{%
4066     \ifdefempty\collargs@make@other@groups{}{%

```

The user has executed `\collargsBraces`. We first apply that setting by executing macro `\collargs@make@other@groups`, and then disable our automatic setting of the primary grouping characters.

```

4067 \collargs@make@other@groups
4068 \def\collargs@make@other@groups{%
4069 \let\collargs@make@other@bgroup\@gobble
4070 \let\collargs@make@other@egroup\@gobble
4071 }%

```

Initialize the list of current comment characters. Each entry is of form `\collargs@do{⟨character code⟩}`. The definition must be global, because the macro will be used only once we exit the current group (by `\collargs@fix@cc@from@other@comment`, if at all).

```

4072 \gdef\collargs@comments{%
4073 \let\collargs@do\collargs@make@verbatim@char
4074 \expandafter\collargs@forranges\expandafter{\collargs@verbatim@ranges}%
4075 }
4076 \def\collargs@make@verbatim@char#1{%

```

Store the current category code of the current character.

```

4077 \ifnum\catcode#1=12
4078 \else
4079 \csedef{collargs@cc@#1}{\the\catcode#1}%
4080 \fi
4081 \ifnum\catcode#1=1
4082 \collargs@make@verbatim@bgroup{#1}%
4083 \else
4084 \ifnum\catcode#1=2
4085 \collargs@make@verbatim@egroup{#1}%
4086 \else
4087 \ifnum\catcode#1=14
4088 \collargs@make@verbatim@comment{#1}%
4089 \fi

```

Change the category code of the current character (including the comment characters).

```

4090 \ifnum\catcode#1=12
4091 \else
4092 \catcode#1=12\relax
4093 \fi
4094 \fi
4095 \fi
4096 }
4097 \fi

```

`\collargs@make@verbatim@bgroup` This macro changes the category of the opening group character to “other”, but only in the full verbatim mode. Next, it populates `\collargs@bgroups`, to facilitate the potential transition into the other verbatim mode. Finally, it executes `\collargs@make@other@bgroup`, which stores the “other” variant of the current character into `\collargs@other@bgroup`, and automatically disables itself, so that it is only executed for the first encountered opening group character — unless it was already `\relaxed` at the top of `\collargs@make@verbatim` as a consequence of the user executing `\collargsBraces`.

```

4098 \def\collargs@make@verbatim@bgroup#1{%
4099 \ifcollargs@verbatimbraces
4100 \catcode#1=12\relax
4101 \fi
4102 \appto\collargs@bgroups{\collargs@do{#1}}%
4103 \collargs@make@other@bgroup{#1}%
4104 }
4105 \def\collargs@make@other@bgroup#1{%
4106 \collargs@make@char\collargs@other@bgroup{#1}{12}%
4107 \let\collargs@make@other@bgroup\@gobble
4108 }

```


`\collargs@make@verbatim@egroup` Ditto for the closing group character.

```
4109 \def\collargs@make@verbatim@egroup#1{%
4110   \ifcollargs@verbatimbraces
4111     \catcode#1=12\relax
4112   \fi
4113   \appto\collargs@egroups{\collargs@do{#1}}%
4114   \collargs@make@other@egroup{#1}%
4115 }
4116 \def\collargs@make@other@egroup#1{%
4117   \collargs@make@char\collargs@other@egroup{#1}{12}%
4118   \let\collargs@make@other@egroup\@gobble
4119 }
```

`\collargs@make@verbatim@comment` This macro populates `\collargs@make@comments@other`.

```
4120 \def\collargs@make@verbatim@comment#1{%
4121   \gappto\collargs@comments{\collargs@do{#1}}%
4122 }
```

`\collargs@make@no@verbatim` This macro switches back to the non-verbatim mode: in LuaTeX, by switching to the original catcode table; in other engines, by recalling the stored category codes.

```
4123 \ifdefined\luatexversion
4124   \def\collargs@make@no@verbatim{%
4125     \catcodetable\collargs@catcodetable@original\relax
4126   }%
4127 \else
4128   \def\collargs@make@no@verbatim{%
4129     \let\collargs@do\collargs@make@no@verbatim@char
4130     \expandafter\collargs@forranges\expandafter{\collargs@verbatim@ranges}%
4131   }
4132 \fi
4133 \def\collargs@make@no@verbatim@char#1{%
```

The original category code of a characted was stored into `\collargs@cc@⟨character code⟩` by `\collargs@make@verbatim`. (We don't use `\collargs@cc`, because we have a number.)

```
4134   \ifcsname collargs@cc@#1\endcsname
4135     \catcode#1=\csname collargs@cc@#1\endcsname\relax
```

We don't have to restore category code 12.

```
4136   \fi
4137 }
```

8.2.6 Transition between the verbatim and the non-verbatim mode

At the transition from verbatim to non-verbatim mode, and vice versa, we sometimes have to fix the category code of the next argument token. This happens when we have an optional argument type in one mode followed by an argument type in another mode, but the optional argument is absent, or when an optional, but absent, verbatim argument is the last argument in the specification. The problem arises because the presence of optional arguments is determined by looking ahead in the input stream; when the argument is absent, this means that we have fixed the category code of the next token. CollArgs addresses this issue by noting the situations where a token receives the wrong category code, and then does its best to replace that token with the same character of the appropriate category code.

`\ifcollargs@fix@requested` This conditional is set, globally, by the optional argument handlers when the argument is in fact absent, and reset in the central loop after applying the fix if necessary.

```
4138 \newif\ifcollargs@fix@requested
```

`\collargs@fix` This macro selects the fixer appropriate to the transition between the previous verbatim mode (determined by `\ifcollargs@last@verbatim` and `\ifcollargs@last@verbatimbraces`) and the current verbatim mode (which is determined by macros `\ifcollargs@verbatim` and `\ifcollargs@verbatimbraces`); if the category code `fix` was not requested (for this, we check `\ifcollargs@fix@requested`), the macro simply executes the next-code given as the sole argument. The name of the fixer macro has the form `\collargs@fix@<last mode>to<current mode>`, where the modes are given by mnemonic codes: `V` = full verbatim, `v` = partial verbatim, and `N` = non-verbatim.

```
4139 \long\def\collargs@fix#1{%
```

Going through `\edef + \unexpanded` avoids doubling the hashes.

```
4140 \edef\collargs@fix@next{\unexpanded{#1}}%
4141 \ifcollargs@fix@requested
4142   \letcs\collargs@action{collargs@fix@%
4143     \ifcollargs@last@verbatim
4144       \ifcollargs@last@verbatimbraces V\else v\fi
4145   \else
4146     N%
4147   \fi
4148   to%
4149   \ifcollargs@verbatim
4150     \ifcollargs@verbatimbraces V\else v\fi
4151   \else
4152     N%
4153   \fi
4154 }%
4155 \else
4156   \let\collargs@action\collargs@fix@next
4157 \fi
4158 \collargs@action
4159 }
```

`\collargs@fix@NtoN` Nothing to do, continue with the next-code.

```
\collargs@fix@vtov
\collargs@fix@VtoV 4160 \def\collargs@fix@NtoN{\collargs@fix@next}
4161 \let\collargs@fix@vtov\collargs@fix@NtoN
4162 \let\collargs@fix@VtoV\collargs@fix@NtoN
```

`\collargs@fix@Ntov` We do nothing for the group tokens; for other tokens, we redirect to `\collargs@fix@NtoV`.

```
4163 \def\collargs@fix@Ntov{%
4164   \futurelet\collargs@temp\collargs@fix@cc@to@other@ii
4165 }
4166 \def\collargs@fix@cc@to@other@ii{%
4167   \ifcat\noexpand\collargs@temp\bgroup
4168     \let\collargs@action\collargs@fix@next
4169   \else
4170     \ifcat\noexpand\collargs@temp\egroup
4171       \let\collargs@action\collargs@fix@next
4172     \else
4173       \let\collargs@action\collargs@fix@NtoV
4174     \fi
4175   \fi
4176   \collargs@action
4177 }
```

`\collargs@fix@NtoV` The only complication here is that we might be in front of a control sequence that was a result of a previous fix in the other direction.

```
4178 \def\collargs@fix@NtoV{%
```

```

4179 \ifcollargs@double@fix
4180   \ifcollargs@in@second@fix
4181   \expandafter\expandafter\expandafter\collargs@fix@NtoV@secondfix
4182   \else
4183   \expandafter\expandafter\expandafter\collargs@fix@NtoV@onemore
4184   \fi
4185 \else
4186   \expandafter\collargs@fix@NtoV@singlefix
4187 \fi
4188 }

```

This is the usual situation of a single fix. We just use `\string` on the next token here (but note that some situations can't be saved: noone can bring a comment back to life, or distinguish a newline and a space)

```

4189 \def\collargs@fix@NtoV@singlefix{%
4190   \expandafter\collargs@fix@next\string
4191 }

```

If this is the first fix of two, we know `#1` is a control sequence, so it is safe to grab it.

```

4192 \def\collargs@fix@NtoV@onemore#1{%
4193   \collargs@do@one@more@fix{%
4194     \expandafter\collargs@fix@next\string#1%
4195   }%
4196 }

```

If this is the second fix of the two, we have to check whether the next token is a control sequence, and if it is, we need to remember it. Afterwards, we redirect to the single-fix.

```

4197 \def\collargs@fix@NtoV@secondfix{%
4198   \if\noexpand\collargs@temp\relax
4199     \expandafter\collargs@fix@NtoV@secondfix@i
4200   \else
4201     \expandafter\collargs@fix@NtoV@singlefix
4202   \fi
4203 }
4204 \def\collargs@fix@NtoV@secondfix@i#1{%
4205   \gdef\collargs@double@fix@cs@ii{#1}%
4206   \collargs@fix@NtoV@singlefix#1%
4207 }

```

`\collargs@fix@vtoN` Do nothing for the grouping tokens, redirect to `\collargs@fix@VtoN` for other tokens.

```

4208 \def\collargs@fix@vtoN{%
4209   \futurelet\collargs@token\collargs@fix@vtoN@i
4210 }
4211 \def\collargs@fix@vtoN@i{%
4212   \ifcat\noexpand\collargs@token\bgroup
4213     \expandafter\collargs@fix@next
4214   \else
4215     \ifcat\noexpand\collargs@token\egroup
4216       \expandafter\expandafter\expandafter\collargs@fix@next
4217     \else
4218       \expandafter\expandafter\expandafter\collargs@fix@VtoN
4219     \fi
4220   \fi
4221 }

```

`\collargs@fix@vtoV` Redirect group tokens to `\collargs@fix@NtoV`, and do nothing for other tokens.

```

4222 \def\collargs@fix@vtoV{%
4223   \futurelet\collargs@token\collargs@fix@vtoV@i

```

```

4224 }
4225 \def\collargs@fix@vtoV@i{%
4226   \ifcat\noexpand\collargs@token\bgroup
4227     \expandafter\collargs@fix@NtoV
4228   \else
4229     \ifcat\noexpand\collargs@token\egroup
4230     \expandafter\expandafter\expandafter\collargs@fix@NtoV
4231   \else
4232     \expandafter\expandafter\expandafter\collargs@fix@next
4233   \fi
4234 \fi
4235 }

```

\collargs@fix@Vtov Redirect group tokens to \collargs@fix@VtoN, and do nothing for other tokens. #1 is surely of category 12, so we can safely grab it.

```

4236 \def\collargs@fix@catcode@of@braces@fromverbatim#1{%
4237   \ifnum\catcode`#1=1
4238     \expandafter\collargs@fix@VtoN
4239     \expandafter#1%
4240   \else
4241     \ifnum\catcode`#1=2
4242       \expandafter\expandafter\expandafter\collargs@fix@cc@VtoN
4243       \expandafter\expandafter\expandafter#1%
4244     \else
4245       \expandafter\expandafter\expandafter\collargs@fix@next
4246     \fi
4247   \fi
4248 }

```

\collargs@fix@VtoN This is the only complicated part. Control sequences and comments (but not grouping characters!) require special attention. We're fine to grab the token right away, as we know it is of category 12.

```

4249 \def\collargs@fix@VtoN#1{%
4250   \ifnum\catcode`#1=0
4251     \expandafter\collargs@fix@VtoN@escape
4252   \else
4253     \ifnum\catcode`#1=14
4254       \expandafter\expandafter\expandafter\collargs@fix@VtoN@comment
4255     \else
4256       \expandafter\expandafter\expandafter\collargs@fix@VtoN@token
4257     \fi
4258   \fi
4259   #1%
4260 }

```

\collargs@fix@VtoN@token We create a new character with the current category code behind the next-code. This works even for grouping characters.

```

4261 \def\collargs@fix@VtoN@token#1{%
4262   \collargs@insert@char\collargs@fix@next{`#1}{\the\catcode`#1}%
4263 }

```

\collargs@fix@VtoN@comment This macro defines a macro which will, when placed at a comment character, remove the tokens until the end of the line. The code is adapted from the TeX.SE answer at tex.stackexchange.com/a/10454/16819 by Bruno Le Floch.

```

4264 \def\collargs@defcommentstripper#1#2{%

```

We chuck a parameter into the following definition, to grab the (verbatim) comment character. This is why this macro must be executed precisely before the (verbatim) comment character.

```

4265 \def#1##1{%
4266   \begingroup%
4267   \escapechar=`\%
4268   \catcode\endlinechar=\active%

```

We assign the “other” category code to comment characters. Without this, comment characters behind the first one make trouble: there would be no `^^M` at the end of the line, so the comment stripper would gobble the following line as well; in fact, it would gobble all subsequent lines containing a comment character. We also make sure to change the category code of *all* comment characters, even if there is usually just one.

```

4269   \def\collargs@do####1{\catcode####1=12 }%
4270   \collargs@comments
4271   \csname\string#1\endcsname%
4272 }%
4273 \begingroup%
4274 \escapechar=`\%
4275 \lccode`\~=\endlinechar%
4276 \lowercase{%
4277   \expandafter\endgroup
4278   \expandafter\def\csname\string#1\endcsname##1~%
4279 }f%

```

I have removed `\space` from the end of the following line. We don’t want it for our application.

```

4280   \endgroup#2%
4281 }%
4282 }
4283 \collargs@defcommentstripper\collargs@fix@VtoN@comment{%
4284   \collargs@fix@next
4285 }

```

We don’t need the generator any more.

```

4286 \let\collargs@defcommentstripper\relax

```

`\collargs@fix@VtoN@escape` An escape character of category code 12 is the most challenging — and we won’t get things completely right — as we have swim further down the input stream to create a control sequence. This macro will throw away the verbatim escape character `#1`.

```

4287 \def\collargs@fix@VtoN@escape#1{%
4288   \ifcollargs@double@fix

```

We need to do things in a special way if we’re in the double-fix situation triggered by the previous fixing of a control sequence (probably this very one). In that case, we can’t collect it in the usual way because the entire control sequence is spelled out in verbatim.

```

4289   \expandafter\collargs@fix@VtoN@escape@d
4290   \else

```

This here is the usual situation where the escape character was tokenized verbatim, but the control sequence name itself will be collected (right away) in the non-verbatim regime.

```

4291   \expandafter\collargs@fix@VtoN@escape@i
4292   \fi
4293 }
4294 \def\collargs@fix@VtoN@escape@i{%

```

The sole character forming a control symbol name may be of any category. Temporarily redefining the category codes of the craziest characters allows `\collargs@fix@VtoN@escape@ii` to simply grab the following character.

```

4295   \begingroup

```

```

4296 \catcode`\ =12
4297 \catcode`\{=12
4298 \catcode`\}=12
4299 \catcode`\_ =12
4300 \collargs@fix@VtoN@escape@ii
4301 }

```

The argument is the first character of the control sequence name.

```

4302 \def\collargs@fix@VtoN@escape@ii#1{%
4303   \endgroup
4304   \def\collargs@csname{#1}%

```

Only if #1 is a letter may the control sequence name continue.

```

4305   \ifnum\catcode`#1=11
4306     \expandafter\collargs@fix@VtoN@escape@iii
4307   \else

```

In the case of a control space, we have to throw away the following spaces.

```

4308     \ifnum\catcode`#1=10
4309       \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@s
4310     \else

```

We have a control symbol. That means that we haven't peeked ahead and can thus skip
`\collargs@fix@VtoN@escape@z`.

```

4311       \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@z@i
4312       \fi
4313     \fi
4314 }

```

We still have to collect the rest of the control sequence name. Braces have their usual meaning again, so we have to check for them explicitly (and bail out if we stumble upon them).

```

4315 \def\collargs@fix@VtoN@escape@iii{%
4316   \futurelet\collargs@temp\collargs@fix@VtoN@escape@iv
4317 }
4318 \def\collargs@fix@VtoN@escape@iv{%
4319   \ifcat\noexpand\collargs@temp\bgroup
4320     \let\collargs@action\collargs@fix@VtoN@escape@z
4321   \else
4322     \ifcat\noexpand\collargs@temp\egroup
4323       \let\collargs@action\collargs@fix@VtoN@escape@z
4324     \else
4325       \expandafter\ifx\space\collargs@temp
4326         \let\collargs@action\collargs@fix@VtoN@escape@s
4327       \else
4328         \let\collargs@action\collargs@fix@VtoN@escape@v
4329       \fi
4330     \fi
4331   \fi
4332   \collargs@action
4333 }

```

If we have a letter, store it and loop back, otherwise finish.

```

4334 \def\collargs@fix@VtoN@escape@v#1{%
4335   \ifcat\noexpand#1a%
4336     \appto\collargs@csname{#1}%
4337     \expandafter\collargs@fix@VtoN@escape@iii
4338   \else
4339     \expandafter\collargs@fix@VtoN@escape@z\expandafter#1%
4340   \fi
4341 }

```

Throw away the following spaces.

```

4342 \def\collargs@fix@VtoN@escape@s{%
4343   \futurelet\collargs@temp\collargs@fix@VtoN@escape@s@i
4344 }
4345 \def\collargs@fix@VtoN@escape@s@i{%
4346   \expandafter\ifx\space\collargs@temp
4347     \expandafter\collargs@fix@VtoN@escape@s@ii
4348   \else
4349     \expandafter\collargs@fix@VtoN@escape@z
4350   \fi
4351 }
4352 \def\collargs@fix@VtoN@escape@s@ii{%
4353   \expandafter\collargs@fix@VtoN@escape@z\romannumeral-0%
4354 }

```

Once we have collected the control sequence name into `\collargs@csname`, we will create the control sequence behind the next-code. However, we have two complications. The minor one is that `\csname` defines an unexisting control sequence to mean `\relax`, so we have to check whether the control sequence we will create is defined, and if not, “undefine” it in advance.

```

4355 \def\collargs@fix@VtoN@escape@z@i{%
4356   \collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin
4357   \collargs@fix@VtoN@escape@z@ii
4358 }%
4359 \def\collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin{%
4360   \ifcsname\collargs@csname\endcsname
4361     \@tempswatrue
4362   \else
4363     \@tempswafalse
4364   \fi
4365 }
4366 \def\collargs@fix@VtoN@escape@z@maybe@undefine@cs@end{%
4367   \if@tempswa
4368   \else
4369     \cslet{\collargs@csname}\collargs@undefined
4370   \fi
4371 }
4372 \def\collargs@fix@VtoN@escape@z@ii{%
4373   \expandafter\collargs@fix@VtoN@escape@z@maybe@undefine@cs@end
4374   \expandafter\collargs@fix@next\csname\collargs@csname\endcsname
4375 }

```

The second complication is much greater, but it only applies to control words and spaces, and that’s why control symbols went directly to the macro above. Control words and spaces will only get there via a detour through the following macro.

The problem is that collecting the control word/space name peeked ahead in the stream, so the character following the control sequence (name) is already tokenized. We will (at least partially) address this by requesting a “double-fix”: until the control sequence we’re about to create is consumed into some argument, each category code fix will fix two “tokens” rather than one.

```

4376 \def\collargs@fix@VtoN@escape@z{%
4377   \collargs@if@one@more@fix{%

```

Some previous fixing has requested a double fix, so let’s do it. Afterwards, redirect to the control symbol code `\collargs@fix@VtoN@escape@z@i`. It will surely use the correct `\collargs@csname` because we do the second fix in a group.

```

4378     \collargs@do@one@more@fix\collargs@fix@VtoN@escape@z@i
4379   }-{}%

```


Remember the collected control sequence. It will be used in `\collargs@cancel@double@fix`.

```
4380 \collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin
4381 \xdef\collargs@double@fix@cs@i{\expandonce{\csname\collargs@csname\endcsname}}%
4382 \collargs@fix@VtoN@escape@z@maybe@undefine@cs@end
```

Request the double-fix.

```
4383 \global\collargs@double@fixtrue
```

The complication is addressed, redirect to the control symbol finish.

```
4384 \collargs@fix@VtoN@escape@z@ii
4385 }%
4386 }
```

When we have to “redo” a control sequence, because it was ping-ponged back into the verbatim mode, we cannot collect it by `\collargs@fix@VtoN@escape@i`, because it is spelled out entirely in verbatim. However, we have seen this control sequence before, and remembered it, so we’ll simply grab it. Another complication is that we might be either at the “first” control sequence, whose fixing created all these double-fix trouble, or at the “second” control sequence, if the first one was immediately followed by another one. But we have remembered both of them: the first one in `\collargs@fix@VtoN@escape@z`, the second one in `\collargs@fix@NtoV@secondfix`.

```
4387 \def\collargs@fix@VtoN@escape@d{%
4388 \ifcollargs@in@second@fix
4389 \expandafter\collargs@fix@VtoN@escape@d@i
4390 \expandafter\collargs@double@fix@cs@ii
4391 \else
4392 \expandafter\collargs@fix@VtoN@escape@d@i
4393 \expandafter\collargs@double@fix@cs@i
4394 \fi
4395 }
```

We have the contents of either `\collargs@double@fix@cs@i` or `\collargs@double@fix@cs@ii` here, a control sequence in both cases.

```
4396 \def\collargs@fix@VtoN@escape@d@i#1{%
4397 \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@d@ii
4398 \expandafter\string#1\relax
4399 }
```

We have the verbatimized control sequence name in #2 (#1 is the escape character). By storing it into `\collargs@csname`, we pretend we have collected it. By defining and executing `\collargs@fix@VtoN@escape@d@iii`, we actually gobble it from the input stream. Finally, we reroute to `\collargs@fix@VtoN@escape@z`.

```
4400 \def\collargs@fix@VtoN@escape@d@ii#1#2\relax{%
4401 \def\collargs@csname{#2}%
4402 \def\collargs@fix@VtoN@escape@d@iii#2{%
4403 \collargs@fix@VtoN@escape@z
4404 }%
4405 \collargs@fix@VtoN@escape@d@iii
4406 }
```

This conditional signals a double-fix request. It should be always set globally, because it is cleared by `\collargs@double@fixfalse` in a group.

```
4407 \newif\ifcollargs@double@fix
```

This conditional signals that we’re currently performing the second fix.

```
4408 \newif\ifcollargs@in@second@fix
```

Inspect the two conditionals above to decide whether we have to perform another fix: if so, execute the first argument, otherwise the second one. This macro is called only from `\collargs@fix@VtoN@escape@z` and `\collargs@fix@NtoV`, because these are the only two places where we might need the second fix, ping-ponging a control sequence between the verbatim and the non-verbatim mode.

```

4409 \def\collargs@if@one@more@fix{%
4410   \ifcollargs@double@fix
4411     \ifcollargs@in@second@fix
4412       \expandafter\expandafter\expandafter\@secondoftwo
4413     \else
4414       \expandafter\expandafter\expandafter\@firstoftwo
4415     \fi
4416   \else
4417     \expandafter\@secondoftwo
4418   \fi
4419 }
4420 \def\collargs@do@one@more@fix#1{%

```

We perform the second fix in a group, signalling that we’re performing it.

```

4421   \begingroup
4422   \collargs@in@second@fixtrue

```

Reexecute the fixing routine, at the end, close the group and execute the given code afterwards.

```

4423   \collargs@fix{%
4424     \endgroup
4425     #1%
4426   }%
4427 }

```

This macro is called from `\collargs@appendarg` to cancel the double-fix request.

```

4428 \def\collargs@cancel@double@fix{%

```

`\collargs@appendarg` is only executed when something was actually consumed. We thus know that at least one of the problematic “tokens” is gone, so the double fix is not necessary anymore.

```

4429   \global\collargs@double@fixfalse

```

What we have to figure out, still, is whether both problematic “tokens” we consumed. If so, no more fixing is required. But if only one of them was consumed, we need to request the normal, single, fix for the remaining “token”.

```

4430   \begingroup

```

This will attach the delimiters directly to the argument, so we’ll see what was actually consumed.

```

4431   \collargs@process@arg

```

We compare what was consumed when collecting the current argument with the control word that triggered double-fixing. If they match, only the offending control word was consumed, so we need to set the fix request to true for the following token.

```

4432   \edef\collargs@temp{\the\collargsArg}%
4433   \edef\collargs@tempa{\expandafter\string\collargs@double@fix@cs@i}%
4434   \ifx\collargs@temp\collargs@tempa
4435     \global\collargs@fix@requestedtrue
4436   \fi
4437   \endgroup
4438 }

```

`\collargs@insert@char` These macros create a character of character code #2 and category code #3. The first macro `\collargs@make@char` inserts it into the stream behind the code in #1; the second one defines the control sequence in #1 to hold the created character (clearly, it should not be used for categories 1 and 2).

We use the facilities of Lua_{TEX}, X_Y_{TEX} and L^A_{TEX} where possible. In the end, we only have to implement our own macros for plain pdf_{TEX}.

```

4439 (!context) \ifdefined\luatexversion
4440   \def\collargs@insert@char#1#2#3{%
4441     \edef\collargs@temp{\unexpanded{#1}}%
4442     \expandafter\collargs@temp\directlua{%
4443       tex.cprint(\number#3,string.char(\number#2))}%
4444   }%
4445   \def\collargs@make@char#1#2#3{%
4446     \edef#1{\directlua{tex.cprint(\number#3,string.char(\number#2))}}%
4447   }%
4448 (*!context)
4449 \else
4450   \ifdefined\XeTeXversion
4451     \def\collargs@insert@char#1#2#3{%
4452       \edef\collargs@temp{\unexpanded{#1}}%
4453       \expandafter\collargs@temp\Ucharcat #2 #3
4454     }%
4455     \def\collargs@make@char#1#2#3{%
4456       \edef#1{\Ucharcat#2 #3}%
4457     }%
4458   \else
4459     (*!latex)
4460     \ExplSyntaxOn
4461     \def\collargs@insert@char#1#2#3{%
4462       \edef\collargs@temp{\unexpanded{#1}}%
4463       \expandafter\expandafter\expandafter\collargs@temp\char_generate:nn{#2}{#3}%
4464     }%
4465     \def\collargs@make@char#1#2#3{%
4466       \edef#1{\char_generate:nn{#2}{#3}}%
4467     }%
4468     \ExplSyntaxOff
4469     (/!latex)
4470     (*!plain)

```

The implementation is inspired by `expl3`'s implementation of `\char_generate:nn`, but our implementation is not expandable, for simplicity. We first store an (arbitrary) character `^^@` of category code `n` into control sequence `\collargs@charofcat@n`, for every (implementable) category code.

```

4471   \begingroup
4472   \catcode`^^@=1 \csgdef{collargs@charofcat@1}{%
4473     \noexpand\expandafter^^@\iffalse}\fi}
4474   \catcode`^^@=2 \csgdef{collargs@charofcat@2}{\iffalse{\fi^^@}}
4475   \catcode`^^@=3 \csgdef{collargs@charofcat@3}{^^@}
4476   \catcode`^^@=4 \csgdef{collargs@charofcat@4}{^^@}

```

As we have grabbed the spaces already, a remaining newline should surely be fixed into a `\par`.

```

4477   \csgdef{collargs@charofcat@5}{\par}
4478   \catcode`^^@=6 \csxdef{collargs@charofcat@6}{\unexpanded{^^@}}
4479   \catcode`^^@=7 \csgdef{collargs@charofcat@7}{^^@}
4480   \catcode`^^@=8 \csgdef{collargs@charofcat@8}{^^@}
4481   \csgdef{collargs@charofcat@10}{\noexpand\space}
4482   \catcode`^^@=11 \csgdef{collargs@charofcat@11}{^^@}
4483   \catcode`^^@=12 \csgdef{collargs@charofcat@12}{^^@}
4484   \catcode`^^@=13 \csgdef{collargs@charofcat@13}{^^@}
4485   \endgroup
4486   \def\collargs@insert@char#1#2#3{%

```

Temporarily change the lowercase code of $\sim\sim\sim$ to the requested character #2.

```
4487 \begingroup
4488 \lccode`~\sim=#2\relax
```

We'll have to close the group before executing the next-code.

```
4489 \def\collargs@temp{\endgroup#1}%
```

$\collargs@charofcat@$ *requested category code* is f-expanded first, leaving us to lowercase $\expandafter\collargs@temp\sim\sim\sim$. Clearly, lowercasing $\expandafter\collargs@temp$ is a no-op, but lowercasing $\sim\sim\sim$ gets us the requested character of the requested category. \expandafter is executed next, and this gets rid of the conditional for category codes 1 and 2.

```
4490 \expandafter\lowercase\expandafter{%
4491 \expandafter\expandafter\expandafter\collargs@temp
4492 \romannumeral-`0\csname collargs@charofcat@the\numexpr#3\relax\endcsname
4493 }%
4494 }
```

This macro cannot not work for category code 6 (because we assign the result to a macro), but no matter, we only use it for category code 12 anyway.

```
4495 \def\collargs@make@char#1#2#3{%
4496 \begingroup
4497 \lccode`~\sim=#2\relax
```

Define $\collargs@temp$ to hold $\sim\sim\sim$ of the appropriate category.

```
4498 \edef\collargs@temp{%
4499 \csname collargs@charofcat@the\numexpr#3\relax\endcsname}%
```

Preexpand the second $\collargs@temp$ so that we lowercase $\def\collargs@temp{\sim\sim\sim}$, with $\sim\sim\sim$ of the appropriate category.

```
4500 \expandafter\lowercase\expandafter{%
4501 \expandafter\def\expandafter\collargs@temp\expandafter{\collargs@temp}%
4502 }%
4503 \expandafter\endgroup
4504 \expandafter\def\expandafter#1\expandafter{\collargs@temp}%
4505 }
4506 </plain>
4507 \fi
4508 \fi
4509 </!context>
```

```
4510 <plain>\resetatcatcode
4511 <context>\stopmodule
4512 <context>\protect
```

Local Variables: TeX-engine: luatex TeX-master: "doc/memoize-code.tex" TeX-auto-save:
nil End:

9 The scripts

9.1 The Perl extraction script `memoize-extract.pl`

```
4513 my $PROG = 'memoize-extract.pl';
4514 my $VERSION = '2024/03/15 v1.2.0';
4515
4516 use strict;
4517 use File::Basename qw/basename/;
4518 use Getopt::Long;
```

```

4519 use File::Spec::Functions
4520     qw/splitpath catpath splitdir rootdir file_name_is_absolute/;
4521 use File::Path qw(make_path);

```

We will only try to import the PDF processing library once we set up the error log. Declare variables for command-line arguments and for `kpathsea` variables. They are defined here so that they are global in the subs which use them.

```

4522 our ($pdf_file, $prune, $keep, $format, $force, $quiet,
4523     $pdf_library, $print_version, $mkdir, $help,
4524     $openin_any, $openout_any, $texmfoutput, $texmf_output_directory);

```

Messages The messages are written both to the extraction log and the terminal (we output to stdout rather than stderr so that messages on the TeX terminal and document `.log` appear in chronological order). Messages are automatically adapted to the TeX `--format`. The format of the messages. It depends on the given `--format`; the last entry is for t the terminal output.

```

4525 my %ERROR = (
4526     latex => '\PackageError{memoize (perl-based extraction)}{$short}{$long}',
4527     plain => '\errhelp{$long}\errmessage{memoize (perl-based extraction): $short}',
4528     context => '\errhelp{$long}\errmessage{memoize (perl-based extraction): $short}',
4529     '' => '$header$short. $long');
4530
4531 my %WARNING = (
4532     latex => '\PackageWarning{memoize (perl-based extraction)}{$texindent$text}',
4533     plain => '\message{memoize (perl-based extraction) Warning: $texindent$text}',
4534     context => '\message{memoize (perl-based extraction) Warning: $texindent$text}',
4535     '' => '$header$indent$text.');
4536
4537 my %INFO = (
4538     latex => '\PackageInfo{memoize (perl-based extraction)}{$texindent$text}',
4539     plain => '\message{memoize (perl-based extraction): $texindent$text}',
4540     context => '\message{memoize (perl-based extraction): $texindent$text}',
4541     '' => '$header$indent$text.');

```

Some variables used in the message routines; note that `header` will be redefined once we parse the arguments.

```

4542 my $exit_code = 0;
4543 my $log;
4544 my $header = '';
4545 my $indent = '';
4546 my $texindent = '';

```

The message routines.

```

4547 sub error {
4548     my ($short, $long) = @_;
4549     if (! $quiet) {
4550         $_ = $ERROR{' '};
4551         s/\$header/\$header/;
4552         s/\$short/\$short/;
4553         s/\$long/\$long/;
4554         print(STDOUT "$_\n");
4555     }
4556     if ($log) {
4557         $long =~ s/\\/\\string\\/g;
4558         $_ = $ERROR{$format};
4559         s/\$short/\$short/;
4560         s/\$long/\$long/;
4561         print(LOG "$_\n");
4562     }
4563     $exit_code = 11;
4564     endinput();
4565 }
4566

```

```

4567 sub warning {
4568     my $text = shift;
4569     if ($log) {
4570         $_ = $WARNING{$format};
4571         s/\$texindent/$texindent/;
4572         s/\$text/$text/;
4573         print(LOG "$_\n");
4574     }
4575     if (! $quiet) {
4576         $_ = $WARNING{' '};
4577         s/\$header/$header/;
4578         s/\$indent/$indent/;
4579         s/\$text/$text/;
4580         print(STDOUT "$_\n");
4581     }
4582     $exit_code = 10;
4583 }
4584
4585 sub info {
4586     my $text = shift;
4587     if ($text && ! $quiet) {
4588         $_ = $INFO{' '};
4589         s/\$header/$header/;
4590         s/\$indent/$indent/;
4591         s/\$text/$text/;
4592         print(STDOUT "$_\n");
4593         if ($log) {
4594             $_ = $INFO{$format};
4595             s/\$texindent/$texindent/;
4596             s/\$text/$text/;
4597             print(LOG "$_\n");
4598         }
4599     }
4600 }

```

Mark the log as complete and exit.

```

4601 sub endinput {
4602     if ($log) {
4603         print(LOG "\\endinput\n");
4604         close(LOG);
4605     }
4606     exit $exit_code;
4607 }
4608
4609 sub die_handler {
4610     stderr_to_warning();
4611     my $text = shift;
4612     chomp($text);
4613     error("Perl error: $text", ' ');
4614 }
4615
4616 sub warn_handler {
4617     my $text = shift;
4618     chomp($text);
4619     warning("Perl warning: $text");
4620 }

```

This is used to print warning messages from PDF::Builder, which are output to STDERR.

```

4621 my $stderr;
4622 sub stderr_to_warning {
4623     if ($stderr) {
4624         my $w = ' Perl info: ';
4625         my $nl = ' ';

```

```

4626         for (split(/\n/, $stderr)) {
4627             /^(^s*)(.*?)(\s*)$/;
4628             $w .= ($1 ? ' ' : $nl) . $2;
4629             $nl = "\n";
4630         }
4631         warning("$w");
4632         $stderr = '';
4633     }
4634 }

```

[Permission-related functions](#) We will need these variables below. Note that we only support Unix and Windows.

```

4635 my $on_windows = $^O eq 'MSWin32';
4636 my $dirsep = $on_windows ? '\\' : '/';

```

`paranoia_in/out` should work exactly as `kpsewhich -safe-in-name/-safe-out-name`.

```

4637 sub paranoia_in {
4638     my ($f, $remark) = @_;
4639     error("I'm not allowed to read from '$f' (openin_any = $openin_any)",
4640         $remark) unless _paranoia($f, $openin_any);
4641 }
4642
4643 sub paranoia_out {
4644     my ($f, $remark) = @_;
4645     error("I'm not allowed to write to '$f' (openin_any = $openout_any)",
4646         $remark) unless _paranoia($f, $openout_any);
4647 }
4648
4649 sub _paranoia {

```

`f` is the path to the file (it should not be empty), and `mode` is the value of `openin_any` or `openout_any`.

```

4650     my ($f, $mode) = @_;
4651     return if (! $f);

```

We split the filename into the directory and the basename part, and the directory into components.

```

4652     my ($volume, $dir, $basename) = splitpath($f);
4653     my @dir = splitdir($dir);
4654     return (

```

In mode ‘any’ (a, y or 1), we may access any file.

```

4655         $mode =~ /^[ay1]$/
4656         || (

```

Otherwise, we are at least in the restricted mode, so we should not open dot files on Unix-like systems (except file called `.tex`).

```

4657         ! (!$on_windows && $basename =~ /\./ && !($basename =~ /\.\tex$/))
4658         && (

```

If we are precisely in the restricted mode (r, n, 0), then there are no further restrictions.

```

4659         $mode =~ /^[rn0]$/

```

Otherwise, we are in the paranoid mode (officially p, but any other value is interpreted as p as well). There are two further restrictions in the paranoid mode.

```

4660         || (

```

We’re not allowed to go to a parent directory.

```

4661             ! grep(/^\.\/$/, @dir) && $basename ne '...'
4662             && (

```

If the given path is absolute, it should be a descendant of either `TEXMF_OUTPUT_DIRECTORY` or `TEXMFOUTPUT`.

```

4663                 (!file_name_is_absolute($f)
4664                 ||

```



```

4665             is_ancestor($texmf_output_directory, $f)
4666             ||
4667             is_ancestor($texmfoutput, $f)
4668         ))));
4669 }

```

Only removes final `"/`s. This is unlike `File::Spec's canonpath`, which also removes `.` components, collapses multiple `/` — and unfortunately also goes up for `..` on Windows.

```

4670 sub normalize_path {
4671     my $path = shift;
4672     my ($v, $d, $n) = splitpath($path);
4673     if ($n eq '' && $d =~ /^[^\\Q$dirsep\E]\\Q$dirsep\E+$/ ) {
4674         $path =~ s/\\Q$dirsep\E+$/ /;
4675     }
4676     return $path;
4677 }

```

On Windows, we disallow “semi-absolute” paths, i.e. paths starting with the `\` but lacking the drive. `File::Spec's file_name_is_absolute` returns 2 if the path is absolute with a volume, 1 if it's absolute with no volume, and 0 otherwise. After a path was sanitized using this function, `file_name_is_absolute` will work as we want it to.

```

4678 sub sanitize_path {
4679     my $f = normalize_path(shift);
4680     my ($v, $d, $n) = splitpath($f);
4681     if ($on_windows) {
4682         my $a = file_name_is_absolute($f);
4683         if ($a == 1 || ($a == 0 && $v) ) {
4684             error("\Semi-absolute\" paths are disallowed: " . $f,
4685                 "The path must either both contain the drive letter and " .
4686                 "start with '\\', or none of these; paths like 'C:foo\\bar' " .
4687                 "and '\\foo\\bar' are disallowed");
4688         }
4689     }
4690 }
4691
4692 sub access_in {
4693     return -r shift;
4694 }
4695
4696 sub access_out {
4697     my $f = shift;
4698     my $exists;
4699     eval { $exists = -e $f };

```

Presumably, we get this error when the parent directory is not executable.

```

4700     return if ($@);
4701     if ($exists) {

```

An existing file should be writable, and if it's a directory, it should also be executable.

```

4702         my $rw = -w $f; my $rd = -d $f; my $rx = -x $f;
4703         return -w $f && (! -d $f || -x $f);
4704     } else {

```

For a non-existing file, the parent directory should be writable. (This is the only place where function `parent` is used, so it's ok that it returns the logical parent.)

```

4705         my $p = parent($f);
4706         return -w $p;
4707     }
4708 }

```

This function finds the location for an input file, respecting `TEXMF_OUTPUT_DIRECTORY` and `TEXMFOUTPUT`, and the permissions in the filesystem. It returns an absolute file as-is. For a

relative file, it tries `TEXMF_OUTPUT_DIRECTORY` (if defined), the current directory (always), and `TEXMFOUTPUT` directory (if defined), in this order. The first readable file found is returned; if no readable file is found, the file in the current directory is returned.

```

4709 sub find_in {
4710     my $f = shift;
4711     sanitize_path($f);
4712     return $f if file_name_is_absolute($f);
4713     for my $df (
4714         $texmf_output_directory ? join_paths($texmf_output_directory, $f) : undef,
4715         $f,
4716         $texmfoutput ? join_paths($texmfoutput, $f) : undef) {
4717         return $df if $df && -r $df;
4718     }
4719     return $f;
4720 }

```

This function finds the location for an output file, respecting `TEXMF_OUTPUT_DIRECTORY` and `TEXMFOUTPUT`, and the permissions in the filesystem. It returns an absolute file as-is. For a relative file, it tries `TEXMF_OUTPUT_DIRECTORY` (if defined), the current directory (unless `TEXMF_OUTPUT_DIRECTORY` is defined), and `TEXMFOUTPUT` directory (if defined), in this order. The first writable file found is returned; if no writable file is found, the file in either the current or the output directory is returned.

```

4721 sub find_out {
4722     my $f = shift;
4723     sanitize_path($f);
4724     return $f if file_name_is_absolute($f);
4725     for my $df (
4726         $texmf_output_directory ? join_paths($texmf_output_directory, $f) : undef,
4727         $texmf_output_directory ? undef : $f,
4728         $texmfoutput ? join_paths($texmfoutput, $f) : undef) {
4729         return $df if $df && access_out($df);
4730     }
4731     return $texmf_output_directory ? join_paths($texmf_output_directory, $f) : $f;
4732 }

```

We next define some filename-related utilities matching what Python offers out of the box. We avoid using `File::Spec`'s `canonpath`, because on Windows, which has no concept of symlinks, this function resolves `..` to the parent.

```

4733 sub name {
4734     my $path = shift;
4735     my ($volume, $dir, $filename) = splitpath($path);
4736     return $filename;
4737 }
4738
4739 sub suffix {
4740     my $path = shift;
4741     my ($volume, $dir, $filename) = splitpath($path);
4742     $filename =~ /\.[^.]*$/;
4743     return $&;
4744 }
4745
4746 sub with_suffix {
4747     my ($path, $suffix) = @_ ;
4748     my ($volume, $dir, $filename) = splitpath($path);
4749     if ($filename =~ s/\.[^.]*$/$suffix/) {
4750         return catpath($volume, $dir, $filename);
4751     } else {
4752         return catpath($volume, $dir, $filename . $suffix);
4753     }
4754 }
4755
4756 sub with_name {

```

```

4757     my ($path, $name) = @_;
4758     my ($volume, $dir, $filename) = splitpath($path);
4759     my ($v,$d,$f) = splitpath($name);
4760     die "Runtime error in with_name: " .
4761 "'$name' should not contain the directory component"
4762     unless $v eq '' && $d eq '' && $f eq $name;
4763     return catpath($volume, $dir, $name);
4764 }
4765
4766 sub join_paths {
4767     my $path1 = normalize_path(shift);
4768     my $path2 = normalize_path(shift);
4769     return $path2 if !$path1 || file_name_is_absolute($path2);
4770     my ($volume1, $dir1, $filename1) = splitpath($path1, 'no_file');
4771     my ($volume2, $dir2, $filename2) = splitpath($path2);
4772     die if $volume2;
4773     return catpath($volume1,
4774                    join($dirsep, ($dir1 eq $dirsep ? '' : $dir1, $dir2)),
4775                    $filename2);
4776 }

```

The logical parent. The same as `pathlib.parent` in Python.

```

4777 sub parent {
4778     my $f = normalize_path(shift);
4779     my ($v, $dn, $_dummy) = splitpath($f, 1);
4780     my $p_dn = $dn =~ s/[^\Q$dirsep\E]+$//r;
4781     if ($p_dn eq '') {
4782         $p_dn = $dn =~ /\^\Q$dirsep\E/ ? $dirsep : '.';
4783     }
4784     my $p = catpath($v, $p_dn, '');
4785     $p = normalize_path($p);
4786     return $p;
4787 }

```

This function assumes that both paths are absolute; ancestor may be "", signaling a non-path.

```

4788 sub is_ancestor {
4789     my $ancestor = normalize_path(shift);
4790     my $descendant = normalize_path(shift);
4791     return if !$ancestor;
4792     $ancestor .= $dirsep unless $ancestor =~ /\Q$dirsep\E$/;
4793     return $descendant =~ /\^\Q$ancestor/;
4794 }

```

A paranoid `Path.mkdir`. The given folder is preprocessed by `find_out`.

```

4795 sub make_directory {
4796     my $folder = find_out(shift);
4797     if (! -d $folder) {
4798         paranoia_out($folder);

```

Using `make_path` is fine because we know that `TEXMF_OUTPUT_DIRECTORY/TEXMFOUTPUT`, if given, exists, and that “folder” contains no ...

```

4799     make_path($folder);

```

This does not get logged when the function is invoked via `--mkdir`, as it is not clear what the log name should be.

```

4800     info("Created directory $folder");
4801 }
4802 }
4803
4804 sub unquote {
4805     shift =~ s/"(.*)"\/\1/rg;
4806 }

```

[Kpathsea](#) Get the values of `openin_any`, `openout_any`, `TEXMFOUTPUT` and `TEXMF_OUTPUT_DIRECTORY`.

```
4807 my $maybe_backslash = $on_windows ? '\\' : '\\';
4808 my $query = 'kpsewhich -expand-var=' .
4809     "openin_any=$maybe_backslash\$openin_any," .
4810     "openout_any=$maybe_backslash\$openout_any," .
4811     "TEXMFOUTPUT=$maybe_backslash\$TEXMFOUTPUT";
4812 my $kpsewhich_output = `$query`;
4813 if (! $kpsewhich_output) {
```

No TeX? (Note that `kpsewhich` should exist in MiKTeX as well.) In absence of `kpathsea` information, we get very paranoid.

```
4814     ($openin_any, $openout_any) = ('p', 'p');
4815     ($texmfoutput, $texmf_output_directory) = ('', '');
```

Unfortunately, this warning can't make it into the log. But then again, the chances of a missing `kpsewhich` are very slim, and its absence would show all over the place anyway.

```
4816     warning('I failed to execute "kpsewhich", is there no TeX system installed? ' .
4817         'Assuming openin_any = openout_any = "p" ' .
4818         '(i.e. restricting all file operations to non-hidden files ' .
4819         'in the current directory of its subdirectories).');
4820 } else {
4821     $kpsewhich_output =~ /^openin_any=(.*/),openout_any=(.*/),TEXMFOUTPUT=(.*/);
4822     ($openin_any, $openout_any, $texmfoutput) = @{^CAPTURE};
4823     $texmf_output_directory = $ENV{'TEXMF_OUTPUT_DIRECTORY'};
4824     if ($openin_any =~ '^\$openin_any') {
```

When the `open*_any` variables are not expanded, we assume we're running MiKTeX. The two config settings below correspond to TeXLive's `openin_any` and `openout_any`; afaik, there is no analogue to `TEXMFOUTPUT`.

```
4825     $query = 'initexmf --show-config-value=[Core]AllowUnsafeInputFiles ' .
4826         '--show-config-value=[Core]AllowUnsafeOutputFiles';
4827     my $initexmf_output = `$query`;
4828     $initexmf_output =~ /^(.*)\n(.*)\n/m;
4829     $openin_any = $1 eq 'true' ? 'a' : 'p';
4830     $openout_any = $2 eq 'true' ? 'a' : 'p';
4831     $texmfoutput = '';
4832     $texmf_output_directory = '';
4833 }
4834 }
```

An output directory should exist, and may not point to the root on Linux. On Windows, it may point to the root, because being absolute also implies containing the drive; see `sanitize_filename`.

```
4835 sub sanitize_output_dir {
4836     return unless my $d = shift;
4837     sanitize_path($d);
```

On Windows, `rootdir` returns `\`, so it cannot possibly match `$d`.

```
4838     return $d if -d $d && $d ne rootdir();
4839 }
4840
4841 $texmfoutput = sanitize_output_dir($texmfoutput);
4842 $texmf_output_directory = sanitize_output_dir($texmf_output_directory);
```

We don't delve into the real script when loaded from the testing code.

```
4843 return 1 if caller;
```

Arguments

```
4844 my $usage = "usage: $PROG [-h] [-P PDF] [-p] [-k] [-F {latex,plain,context}] [-f] " .
4845     "[-L {PDF::API2,PDF::Builder}] [-q] [-m] [-V] mmz\n";
4846 my $Help = <<END;
4847 Extract extern pages produced by package Memoize out of the document PDF.
4848
```

```

4849 positional arguments:
4850 mmz                the record file produced by Memoize:
4851                    doc.mmz when compiling doc.tex
4852                    (doc and doc.tex are accepted as well)
4853
4854 options:
4855 -h, --help          show this help message and exit
4856 -P PDF, --pdf PDF   extract from file PDF
4857 -p, --prune         remove the extern pages after extraction
4858 -k, --keep          do not mark externs as extracted
4859 -F, --format {latex,plain,context}
4860                    the format of the TeX document invoking extraction
4861 -f, --force         extract even if the size-check fails
4862 -q, --quiet         describe what's happening
4863 -L, --library {PDF::API2, PDF::Builder}
4864                    which PDF library to use for extraction (default: PDF::API2)
4865 -m, --mkdir         create a directory (and exit);
4866                    mmz argument is interpreted as directory name
4867 -V, --version       show program's version number and exit
4868
4869 For details, see the man page or the Memoize documentation.
4870 END
4871
4872 my @valid_libraries = ('PDF::API2', 'PDF::Builder');
4873 Getopt::Long::Configure ("bundling");
4874 GetOptions(
4875     "pdf|P=s"    => \$pdf_file,
4876     "prune|p"    => \$prune,
4877     "keep|k"     => \$keep,
4878     "format|F=s" => \$format,
4879     "force|f"    => \$force,
4880     "quiet|q"    => \$quiet,
4881     "library|L=s" => \$pdf_library,
4882     "mkdir|m"    => \$mkdir,
4883     "version|V"  => \$print_version,
4884     "help|h|?"  => \$help,
4885 ) or die $usage;
4886
4887 if ($help) {print("$usage\n$Help"); exit 0}
4888
4889 if ($print_version) { print("$PROG of Memoize $VERSION\n"); exit 0 }
4890
4891 die "${usage}$PROG: error: the following arguments are required: mmz\n"
4892     unless @ARGV == 1;
4893
4894 die "${usage}$PROG: error: argument -F/--format: invalid choice: '$format' " .
4895     "(choose from 'latex', 'plain', 'context')\n"
4896     unless grep $_ eq $format, ('', 'latex', 'plain', 'context');
4897
4898 die "${usage}$PROG: error: argument -L/--library: invalid choice: '$pdf_library' " .
4899     "(choose from " . join(", ", @valid_libraries) . ")\n"
4900     if $pdf_library && ! grep $_ eq $pdf_library, @valid_libraries;
4901
4902 $header = $format ? basename($0) . ': ' : '';

```

start a new line in the TeX terminal output

```

4903 print("\n") if $format;

```

Initialization With `--mkdir`, argument `mmz` is interpreted as the directory to create.

```

4904 if ($mkdir) {
4905     make_directory($ARGV[0]);
4906     exit 0;
4907 }

```

Normalize the mmz argument into a .mmz filename.

```
4908 my $mmz_file = $ARGV[0];
4909 $mmz_file = with_suffix($mmz_file, '.mmz')
4910     if suffix($mmz_file) eq '.tex';
4911 $mmz_file = with_name($mmz_file, name($mmz_file) . '.mmz')
4912     if suffix($mmz_file) ne '.mmz';
```

Once we have the .mmz filename, we can open the log.

```
4913 if ($format) {
4914     my $_log = find_out(with_suffix($mmz_file, '.mmz.log'));
4915     paranoia_out($_log);
4916     info("Logging to '$_log'");
4917     $log = $_log;
4918     open LOG, ">$log";
4919 }
```

Now that we have opened the log file, we can try loading the PDF processing library.

```
4920 if ($pdf_library) {
4921     eval "use $pdf_library";
4922     error("Perl module '$pdf_library' was not found",
4923         'Have you followed the instructions in section 1.1 of the manual?')
4924     if ($@);
4925 } else {
4926     for (@valid_libraries) {
4927         eval "use $_";
4928         if (!$@) {
4929             $pdf_library = $_;
4930             last;
4931         }
4932     }
4933     if (!$pdf_library) {
4934         error("No suitable Perl module for PDF processing was found, options are " .
4935             join(", ", @valid_libraries),
4936             'Have you followed the instructions in section 1.1 of the manual?');
4937     }
4938 }
```

Catch any errors in the script and output them to the log.

```
4939 $SIG{__DIE__} = \&die_handler;
4940 $SIG{__WARN__} = \&warn_handler;
4941 close(STDERR);
4942 open(STDERR, ">", \&stderr);
```

Find the .mmz file we will read, but retain the original filename in \$given_mmz_file, as we will still need it.

```
4943 my $given_mmz_file = $mmz_file;
4944 $mmz_file = find_in($mmz_file, 1);
4945 if (! -e $mmz_file) {
4946     info("File '$given_mmz_file' does not exist, assuming there's nothing to do");
4947     endinput();
4948 }
4949 paranoia_in($mmz_file);
4950 paranoia_out($mmz_file,
4951     'I would have to rewrite this file unless option --keep is given.')
4952     unless $keep;
```

Determine the PDF filename: it is either given via --pdf, or constructed from the .mmz filename.

```
4953 $pdf_file = with_suffix($given_mmz_file, '.pdf') if !$pdf_file;
4954 $pdf_file = find_in($pdf_file);
4955 paranoia_in($pdf_file);
4956 paranoia_out($pdf_file,
4957     'I would have to rewrite this file because option --prune was given.')
4958     if $prune;
```

Various initializations.

```
4959 my $pdf;
4960 my %extern_pages;
4961 my $new_mmz;
4962 my $tolerance = 0.01;
4963 info("Extracting new externs listed in '$mmz_file' " .
4964       "from '$pdf_file' using Perl module $pdf_library");
4965 my $done_message = "Done (there was nothing to extract)";
4966 $indent = ' ';
4967 $texindent = '\space\space ';
4968 my $dir_to_make;
```

Process .mmz We cannot process the .mmz file using in-place editing. It would fail when the file is writable but its parent directory is not.

```
4969 open (MMZ, $mmz_file);
4970 while (<MMZ>) {
4971     my $mmz_line = $_;
4972     if (/^\mmzPrefix *{(?P<prefix>)} /) {
```

Found \mmzPrefix: create the extern directory, but only later, if an extern file is actually produced. We parse the prefix in two steps because we have to unquote the entire prefix.

```
4973     my $prefix = unquote($+{prefix});
4974     warning("Cannot parse line '$mmz_line'") unless
4975         $prefix =~ /(P<dir_prefix>.*\/)?(P<name_prefix>.*?)/;
4976     $dir_to_make = $+{dir_prefix};
4977 } elsif (/^\mmzNewExtern\ *{(?P<extern_path>.*?)}{(?P<page_n>[0-9]+)}#
4978         {(P<expected_width>[0-9.]*pt){(P<expected_height>[0-9.]*pt)/x} {
```

Found \mmzNewExtern: extract the extern page into an extern file.

```
4979     $done_message = "Done";
4980     my $ok = 1;
4981     my %m_ne = %+;
```

The extern filename, as specified in .mmz:

```
4982     my $extern_file = unquote($m_ne{extern_path});
```

We parse the extern filename in a separate step because we have to unquote the entire path.

```
4983     warning("Cannot parse line '$mmz_line'") unless
4984         $extern_file =~ /(P<dir_prefix>.*\/)?(P<name_prefix>.*?)#
4985         (P<code_md5sum>[0-9A-F]{32})-#
4986         (P<context_md5sum>[0-9A-F]{32})(?-[0-9]+)?.pdf/x;
```

The actual extern filename:

```
4987     my $extern_file_out = find_out($extern_file);
4988     paranoia_out($extern_file_out);
4989     my $page = $m_ne{page_n};
```

Check whether c-memo and cc-memo exist (in any input directory).

```
4990     my $c_memo = with_name($extern_file,
4991                             $+{name_prefix} . $+{code_md5sum} . '.memo');
4992     my $cc_memo = with_name($extern_file,
4993                             $+{name_prefix} . $+{code_md5sum} .
4994                             '-' . $+{context_md5sum} . '.memo');
4995     my $c_memo_in = find_in($c_memo);
4996     my $cc_memo_in = find_in($cc_memo);
4997     if ((! access_in($c_memo_in) || ! access_in($cc_memo_in)) && !$force) {
4998         warning("I refuse to extract page $page into extern '$extern_file', " .
4999             "because the associated c-memo '$c_memo' and/or " .
5000             "cc-memo '$cc_memo' does not exist");
5001         $ok = '';
5002     }
```

Load the PDF. We only do this now so that we don't load it if there is nothing to extract.

```

5003     if ($ok && ! $pdf) {
5004         if (!access_in($pdf_file)) {
5005             warning("Cannot open '$pdf_file'", '');
5006             endinput();
5007         }

```

Temporarily disable error handling, so that we can catch the error ourselves.

```

5008     $SIG{__DIE__} = undef; $SIG{__WARN__} = undef;

```

All safe, `paranoia_in` was already called above.

```

5009     eval { $pdf = $pdf_library->open($pdf_file, msgver => 0) };
5010     $SIG{__DIE__} = \&die_handler; $SIG{__WARN__} = \&warn_handler;
5011     error("File '$pdf_file' seems corrupted. " .
5012         "Perhaps you have to load Memoize earlier in the preamble",
5013         "In particular, Memoize must be loaded before TikZ library " .
5014         "'fadings' and any package deploying it, and in Beamer, " .
5015         "load Memoize by writing \\RequirePackage{memoize} before " .
5016         "\\documentclass{beamer}." .
5017         "This was the error thrown by Perl:" . "\n$@" ) if $@;
5018 }

```

Does the page exist?

```

5019     if ($ok && $page > (my $n_pages = $pdf->page_count())) {
5020         error("I cannot extract page $page from '$pdf_file', " .
5021             "as it contains only $n_pages page" .
5022             ($n_pages > 1 ? 's' : ''), '');
5023     }
5024     if ($ok) {

```

Import the page into the extern PDF (no disk access yet).

```

5025         my $extern = $pdf_library->new(outver => $pdf->version);
5026         $extern->import_page($pdf, $page);
5027         my $extern_page = $extern->open_page(1);

```

Check whether the page size matches the `.mmz` expectations.

```

5028         my ($x0, $y0, $x1, $y1) = $extern_page->get_mediabox();
5029         my $width_pt = ($x1 - $x0) / 72 * 72.27;
5030         my $height_pt = ($y1 - $y0) / 72 * 72.27;
5031         my $expected_width_pt = $m_ne{expected_width};
5032         my $expected_height_pt = $m_ne{expected_height};
5033         if ((abs($width_pt - $expected_width_pt) > $tolerance
5034             || abs($height_pt - $expected_height_pt) > $tolerance) && !$force) {
5035             warning("I refuse to extract page $page from $pdf_file, " .
5036                 "because its size (${width_pt}pt x ${height_pt}pt) " .
5037                 "is not what I expected " .
5038                 "(${expected_width_pt}pt x ${expected_height_pt}pt)");
5039         } else {

```

All tests were successful, let's create the extern file. First, the containing directory, if necessary.

```

5040         if ($dir_to_make) {
5041             make_directory($dir_to_make);
5042             $dir_to_make = undef;
5043         }

```

Now the extern file. Note that `paranoia_out` was already called above.

```

5044         info("Page $page --> $extern_file_out");
5045         $extern->saveas($extern_file_out);

```

This page will get pruned.

```

5046         $extern_pages{$page} = 1 if $prune;

```

Comment out this `\mmzNewExtern`.

```

5047         $new_mmz .= '%' unless $keep;

```



```

5048         }
5049     }
5050 }
5051 $new_mmz .= $mmz_line unless $keep;
5052 stderr_to_warning();
5053 }
5054 close(MMZ);
5055 $indent = '';
5056 $texindent = '';
5057 info($done_message);

```

Write out the .mmz file with \mmzNewExtern lines commented out. (All safe, paranoia_out was already called above.)

```

5058 if (!$keep) {
5059     open(MMZ, ">", $mmz_file);
5060     print MMZ $new_mmz;
5061     close(MMZ);
5062 }

```

Remove the extracted pages from the original PDF. (All safe, paranoia_out was already called above.)

```

5063 if ($prune and keys(%extern_pages) != 0) {
5064     my $pruned_pdf = $pdf_library->new();
5065     for (my $n = 1; $n <= $pdf->page_count(); $n++) {
5066         if (! $extern_pages{$n}) {
5067             $pruned_pdf->import_page($pdf, $n);
5068         }
5069     }
5070     $pruned_pdf->save($pdf_file);
5071     info("The following extern pages were pruned out of the PDF: " .
5072         join(", ", sort(keys(%extern_pages))));
5073 }
5074
5075 endinput();

```

9.2 The Python extraction script memoize-extract.py

```

5076 __version__ = '2024/03/15 v1.2.0'
5077
5078 import argparse, re, sys, os, subprocess, itertools, traceback, platform
5079 from pathlib import Path, PurePath

```

Messages We will only try to import the PDF processing library once we set up the error log. The messages are written both to the extraction log and the terminal (we output to stdout rather than stderr so that messages on the TeX terminal and document .log appear in chronological order). Messages are automatically adapted to the TeX --format. The format of the messages. It depends on the given --format; the last entry is for the terminal output.

```

5080 ERROR = {
5081     'latex':    r'\PackageError{{{package_name}}}{short}{long}',
5082     'plain':    r'\errhelp{{{long}}}\errmessage{{{package_name}: {short}}}',
5083     'context':  r'\errhelp{{{long}}}\errmessage{{{package_name}: {short}}}',
5084     None:       '{header}{short}.\n{long}',
5085 }
5086
5087 WARNING = {
5088     'latex':    r'\PackageWarning{{{package_name}}}{texindent}{text}',
5089     'plain':    r'\message{{{package_name}: {texindent}{text}}}',
5090     'context':  r'\message{{{package_name}: {texindent}{text}}}',
5091     None:       r'{header}{indent}{text}.',
5092 }
5093
5094 INFO = {
5095     'latex':    r'\PackageInfo{{{package_name}}}{texindent}{text}',

```

```

5096 'plain': r'\message{{{package_name}: {texindent}{text}}}',
5097 'context': r'\message{{{package_name}: {texindent}{text}}}',
5098 None: r'{header}{indent}{text}.',
5099 }

```

Some variables used in the message routines; note that `header` will be redefined once we parse the arguments.

```

5100 package_name = 'memoize (python-based extraction)'
5101 exit_code = 0
5102 log = None
5103 header = ''
5104 indent = ''
5105 texindent = ''

```

The message routines.

```

5106 def error(short, long):
5107     if not args.quiet:
5108         print(ERROR[None].format(short = short, long = long, header = header))
5109     if log:
5110         long = long.replace('\n', '\\string\\n')
5111         print(
5112             ERROR[args.format].format(
5113                 short = short, long = long, package_name = package_name),
5114             file = log)
5115     global exit_code
5116     exit_code = 11
5117     endinput()
5118
5119 def warning(text):
5120     if log:
5121         print(
5122             WARNING[args.format].format(
5123                 text = text, texindent = texindent, package_name = package_name),
5124             file = log)
5125     if not args.quiet:
5126         print(WARNING[None].format(text = text, header = header, indent = indent))
5127     global exit_code
5128     exit_code = 10
5129
5130 def info(text):
5131     if text and not args.quiet:
5132         print(INFO[None].format(text = text, header = header, indent = indent))
5133     if log:
5134         print(
5135             INFO[args.format].format(
5136                 text = text, texindent = texindent, package_name = package_name),
5137             file = log)

```

Mark the log as complete and exit.

```

5138 def endinput():
5139     if log:
5140         print(r'\endinput', file = log)
5141         log.close()
5142     sys.exit(exit_code)

```

[Permission-related functions](#) `paranoia_in/out` should work exactly as `kpsewhich -safe-in-name/-safe-out-name`.

```

5143 def paranoia_in(f, remark = ''):
5144     if f and not _paranoia(f, openin_any):
5145         error(f"I'm not allowed to read from '{f}' (openin_any = {openin_any})",
5146             remark)
5147
5148 def paranoia_out(f, remark = ''):

```

```

5149     if f and not _paranoia(f, openout_any):
5150         error(f"I'm not allowed to write to '{f}' (openout_any = {openout_any})",
5151             remark)
5152
5153 def _paranoia(f, mode):

    mode is the value of openin_any or openout_any. f is a pathlib.Path object.
5154     return (

        In mode 'any' (a, y or 1), we may access any file.
5155         mode in 'ay1'
5156         or (

            Otherwise, we are at least in the restricted mode, so we should not open dot files on Unix-like
            systems (except file called .tex).
5157             not (os.name == 'posix' and f.stem.startswith('.') and f.stem != '.tex')
5158             and (

                If we are precisely in the restricted mode (r, n, 0), then there are no further restrictions.
5159                 mode in 'rn0'

                Otherwise, we are in the paranoid mode (officially p, but any other value is interpreted as p as
                well). There are two further restrictions in the paranoid mode.
5160                 or (

                    We're not allowed to go to a parent directory.
5161                     '..' not in f.parts
5162                     and

                        If the given path is absolute, it should be a descendant of either TEXMF_OUTPUT_DIRECTORY or
                        TEXMFOUTPUT.
5163                         (not f.is_absolute()
5164                         or
5165                         is_ancestor(texmf_output_directory, f)
5166                         or
5167                         is_ancestor(texmfoutput, f)
5168                         ))))

                On Windows, we disallow "semi-absolute" paths, i.e. paths starting with the \ but lacking the
                drive. On Windows, pathlib's is_absolute returns True only for paths starting with \ and
                containing the drive.
5169 def sanitize_filename(f):
5170     if f and platform.system() == 'Windows' and not (f.is_absolute() or not f.drive):
5171         error(f "\"Semi-absolute\" paths are disallowed: '{f}', r\"The path must \"
5172             r\"either contain both the drive letter and start with '\\', \"
5173             r\"or none of these; paths like 'C:foo' and '\\foo' are disallowed\"")
5174
5175 def access_in(f):
5176     return os.access(f, os.R_OK)

    This function can fail on Windows, reporting a non-writable file or dir as writable, because
    os.access does not work with Windows' iclacs permissions. Consequence: we might try to
    write to a read-only current or output directory instead of switching to the temporary directory.
    Paranoia is unaffected, as it doesn't use access_* functions.
5177 def access_out(f):
5178     try:
5179         exists = f.exists()

        Presumably, we get this error when the parent directory is not executable.
5180     except PermissionError:
5181         return
5182     if exists:

```

An existing file should be writable, and if it's a directory, it should also be executable.

```
5183     return os.access(f, os.W_OK) and (not f.is_dir() or os.access(f, os.X_OK))
5184     else:
```

For a non-existing file, the parent directory should be writable. (This is the only place where function `pathlib.parent` is used, so it's ok that it returns the logical parent.)

```
5185     return os.access(f.parent, os.W_OK)
```

This function finds the location for an input file, respecting `TEXMF_OUTPUT_DIRECTORY` and `TEXMFOUTPUT`, and the permissions in the filesystem. It returns an absolute file as-is. For a relative file, it tries `TEXMF_OUTPUT_DIRECTORY` (if defined), the current directory (always), and `TEXMFOUTPUT` directory (if defined), in this order. The first readable file found is returned; if no readable file is found, the file in the current directory is returned.

```
5186 def find_in(f):
5187     sanitize_filename(f)
5188     if f.is_absolute():
5189         return f
5190     for df in (texmf_output_directory / f if texmf_output_directory else None,
5191               f,
5192               texmfoutput / f if texmfoutput else None):
5193         if df and access_in(df):
5194             return df
5195     return f
```

This function finds the location for an output file, respecting `TEXMF_OUTPUT_DIRECTORY` and `TEXMFOUTPUT`, and the permissions in the filesystem. It returns an absolute file as-is. For a relative file, it tries `TEXMF_OUTPUT_DIRECTORY` (if defined), the current directory (unless `TEXMF_OUTPUT_DIRECTORY` is defined), and `TEXMFOUTPUT` directory (if defined), in this order. The first writable file found is returned; if no writable file is found, the file in either the current or the output directory is returned.

```
5196 def find_out(f):
5197     sanitize_filename(f)
5198     if f.is_absolute():
5199         return f
5200     for df in (texmf_output_directory / f if texmf_output_directory else None,
5201               f if not texmf_output_directory else None,
5202               texmfoutput / f if texmfoutput else None):
5203         if df and access_out(df):
5204             return df
5205     return texmf_output_directory / f if texmf_output_directory else f
```

This function assumes that both paths are absolute; ancestor may be `None`, signaling a non-path.

```
5206 def is_ancestor(ancestor, descendant):
5207     if not ancestor:
5208         return
5209     a = ancestor.parts
5210     d = descendant.parts
5211     return len(a) < len(d) and a == d[0:len(a)]
```

A paranoid `Path.mkdir`. The given folder is preprocessed by `find_out`.

```
5212 def mkdir(folder):
5213     folder = find_out(Path(folder))
5214     if not folder.exists():
5215         paranoia_out(folder)
```

Using `folder.mkdir` is fine because we know that `TEXMF_OUTPUT_DIRECTORY/TEXMFOUTPUT`, if given, exists, and that “folder” contains no ...

```
5216     folder.mkdir(parents = True, exist_ok = True)
```

This does not get logged when the function is invoked via `--mkdir`, as it is not clear what the log name should be.

```
5217     info(f"Created directory {folder}")
```

```

5218
5219 _re_unquote = re.compile(r'\"(.*)\"')
5220 def unquote(fn):
5221     return _re_unquote.sub(r'\1', fn)

```

Kpathsea Get the values of `openin_any`, `openout_any`, `TEXMFOUTPUT` and `TEXMF_OUTPUT_DIRECTORY`.

```

5222 kpsewhich_output = subprocess.run(['kpsewhich',
5223                                     f'-expand-var='
5224                                     f'openin_any=$openin_any,',
5225                                     f'openout_any=$openout_any,',
5226                                     f'TEXMFOUTPUT=$TEXMFOUTPUT'],
5227                                     capture_output = True
5228                                     ).stdout.decode().strip()
5229 if not kpsewhich_output:

```

No TeX? (Note that `kpsewhich` should exist in MiKTeX as well.) In absence of `kpathsea` information, we get very paranoid, but still try to get `TEXMFOUTPUT` from an environment variable.

```

5230     openin_any, openout_any = 'p', 'p'
5231     texmfoutput, texmf_output_directory = None, None

```

Unfortunately, this warning can't make it into the log. But then again, the chances of a missing `kpsewhich` are very slim, and its absence would show all over the place anyway.

```

5232     warning('I failed to execute "kpsewhich"; , is there no TeX system installed? '
5233             'Assuming openin_any = openout_any = "p" '
5234             '(i.e. restricting all file operations to non-hidden files '
5235             'in the current directory of its subdirectories).')
5236 else:
5237     m = re.fullmatch(r'openin_any=(.*),openout_any=(.*),TEXMFOUTPUT=(.*)',
5238                     kpsewhich_output)
5239     openin_any, openout_any, texmfoutput = m.groups()
5240     texmf_output_directory = os.environ.get('TEXMF_OUTPUT_DIRECTORY', None)
5241     if openin_any == '$openin_any':

```

When the `open*_any` variables are not expanded, we assume we're running MiKTeX. The two config settings below correspond to TeXLive's `openin_any` and `openout_any`; afaik, there is no analogue to `TEXMFOUTPUT`.

```

5242     initexmf_output = subprocess.run(
5243         ['initexmf', '--show-config-value=[Core]AllowUnsafeInputFiles',
5244          '--show-config-value=[Core]AllowUnsafeOutputFiles'],
5245         capture_output = True).stdout.decode().strip()
5246     openin_any, openout_any = initexmf_output.split()
5247     openin_any = 'a' if openin_any == 'true' else 'p'
5248     openout_any = 'a' if openout_any == 'true' else 'p'
5249     texmfoutput = None
5250     texmf_output_directory = None

```

An output directory should exist, and may not point to the root on Linux. On Windows, it may point to the root, because we only allow absolute filenames containing the drive, e.g. `F:\`; see `is_absolute`.

```

5251 def sanitize_output_dir(d_str):
5252     d = Path(d_str) if d_str else None
5253     sanitize_filename(d)
5254     return d if d and d.is_dir() and \
5255         (not d.is_absolute() or len(d.parts) != 1 or d.drive) else None
5256
5257 texmfoutput = sanitize_output_dir(texmfoutput)
5258 texmf_output_directory = sanitize_output_dir(texmf_output_directory)
5259
5260 class NotExtracted(UserWarning):
5261     pass

```

We don't delve into the real script when loaded from the testing code.

```
5262 if __name__ == '__main__':
```

Arguments

```
5263     parser = argparse.ArgumentParser(
5264         description = "Extract extern pages produced by package Memoize "
5265             "out of the document PDF.",
5266         epilog = "For details, see the man page or the Memoize documentation.",
5267         prog = 'memoize-extract.py',
5268     )
5269     parser.add_argument('-P', '--pdf', help = 'extract from file PDF')
5270     parser.add_argument('-p', '--prune', action = 'store_true',
5271         help = 'remove the extern pages after extraction')
5272     parser.add_argument('-k', '--keep', action = 'store_true',
5273         help = 'do not mark externs as extracted')
5274     parser.add_argument('-F', '--format', choices = ['latex', 'plain', 'context'],
5275         help = 'the format of the TeX document invoking extraction')
5276     parser.add_argument('-f', '--force', action = 'store_true',
5277         help = 'extract even if the size-check fails')
5278     parser.add_argument('-q', '--quiet', action = 'store_true',
5279         help = "describe what's happening")
5280     parser.add_argument('-m', '--mkdir', action = 'store_true',
5281         help = 'create a directory (and exit); '
5282             "'mmz argument is interpreted as directory name'")
5283     parser.add_argument('-V', '--version', action = 'version',
5284         version = f"%(prog)s of Memoize " + __version__)
5285     parser.add_argument('mmz', help = 'the record file produced by Memoize: '
5286         'doc.mmz when compiling doc.tex '
5287         '(doc and doc.tex are accepted as well)')
5288
5289     args = parser.parse_args()
5290
5291     header = parser.prog + ': ' if args.format else ''
```

Start a new line in the TeX terminal output.

```
5292     if args.format:
5293         print()
```

Initialization With `--mkdir`, argument `mmz` is interpreted as the directory to create.

```
5294     if args.mkdir:
5295         mkdir(args.mmz)
5296         sys.exit()
```

Normalize the `mmz` argument into a `.mmz` filename.

```
5297     mmz_file = Path(args.mmz)
5298     if mmz_file.suffix == '.tex':
5299         mmz_file = mmz_file.with_suffix('.mmz')
5300     elif mmz_file.suffix != '.mmz':
5301         mmz_file = mmz_file.with_name(mmz_file.name + '.mmz')
```

Once we have the `.mmz` filename, we can open the log.

```
5302     if args.format:
5303         log_file = find_out(mmz_file.with_suffix('.mmz.log'))
5304         paranoia_out(log_file)
5305         info(f"Logging to '{log_file}");
5306         log = open(log_file, 'w')
```

Now that we have opened the log file, we can try loading the PDF processing library.

```
5307     try:
5308         import pdfwr
5309     except ModuleNotFoundError:
5310         error("Python module 'pdfwr' was not found",
5311             'Have you followed the instructions in section 1.1 of the manual?')
```

Catch any errors in the script and output them to the log.

5312 try:

Find the .mmz file we will read, but retain the original filename in `given_mmz_file`, as we will still need it.

```
5313       given_mmz_file = mmz_file
5314       mmz_file = find_in(mmz_file)
5315       paranoia_in(mmz_file)
5316       if not args.keep:
5317           paranoia_out(mmz_file,
5318                remark = 'This file is rewritten unless option --keep is given.')
5319       try:
5320           mmz = open(mmz_file)
5321       except FileNotFoundError:
5322           info(f"File '{given_mmz_file}' does not exist, "
5323               f"assuming there's nothing to do")
5324       endinput()
```

Determine the PDF filename: it is either given via `--pdf`, or constructed from the .mmz filename.

```
5325       pdf_file = find_in(Path(args.pdf)
5326                           if args.pdf else given_mmz_file.with_suffix('.pdf'))
5327       paranoia_in(pdf_file)
5328       if args.prune:
5329           paranoia_out(pdf_file,
5330                remark = 'I would have to rewrite this file '
5331                'because option --prune was given.')
```

Various initializations.

```
5332       re_prefix = re.compile(r'\\mmzPrefix *{(?P<prefix>.*?)}')
5333       re_split_prefix = re.compile(r'(?P<dir_prefix>.*\/)?(?P<name_prefix>.*?)')
5334       re_newextern = re.compile(
5335           r'\\mmzNewExtern *{(?P<extern_path>.*?)}{(?P<page_n>[0-9]+)}'
5336           r'{(?P<expected_width>[0-9.]*pt){(?P<expected_height>[0-9.]*pt)}')
5337       re_extern_path = re.compile(
5338           r'(?P<dir_prefix>.*\/)?(?P<name_prefix>.*?)'
5339           r'(?P<code_md5sum>[0-9A-F]{32})-'
5340           r'(?P<context_md5sum>[0-9A-F]{32})(?:-[0-9]+)?.pdf')
5341       pdf = None
5342       extern_pages = []
5343       new_mmz = []
5344       tolerance = 0.01
5345       dir_to_make = None
5346       info(f"Extracting new externs listed in '{mmz_file}' from '{pdf_file}'")
5347       done_message = "Done (there was nothing to extract)"
5348       indent = ' '
5349       texindent = '\space\space '
```

Process .mmz

```
5350       for line in mmz:
5351           try:
5352               if m_p := re_prefix.match(line):
```

Found `\mmzPrefix`: create the extern directory, but only later, if an extern file is actually produced. We parse the prefix in two steps because we have to unquote the entire prefix.

```
5353               prefix = unquote(m_p['prefix'])
5354               if not (m_sp := re_split_prefix.match(prefix)):
5355                   warning(f"Cannot parse line {line.strip()}")
5356               dir_to_make = m_sp['dir_prefix']
5357               elif m_ne := re_newextern.match(line):
```

Found `\mmzNewExtern`: extract the extern page into an extern file.

```
5358               done_message = "Done"
```

The extern filename, as specified in .mmz:

```

5359         unquoted_extern_path = unquote(m_ne['extern_path'])
5360         extern_file = Path(unquoted_extern_path)

```

We parse the extern filename in a separate step because we have to unquote the entire path.

```

5361         if not (m_ep := re_extern_path.match(unquoted_extern_path)):
5362             warning(f"Cannot parse line {line.strip()}")

```

The actual extern filename:

```

5363         extern_file_out = find_out(extern_file)
5364         paranoia_out(extern_file_out)
5365         page_n = int(m_ne['page_n'])-1

```

Check whether c-memo and cc-memo exist (in any input directory).

```

5366         c_memo = extern_file.with_name(
5367             m_ep['name_prefix'] + m_ep['code_md5sum'] + '.memo')
5368         cc_memo = extern_file.with_name(
5369             m_ep['name_prefix'] + m_ep['code_md5sum']
5370             + '-' + m_ep['context_md5sum'] + '.memo')
5371         c_memo_in = find_in(c_memo)
5372         cc_memo_in = find_in(cc_memo)
5373         if not (access_in(c_memo_in) and access_in(cc_memo_in)) \
5374             and not args.force:
5375             warning(f"I refuse to extract page {page_n+1} into extern "
5376                     f"'{extern_file}', because the associated c-memo "
5377                     f"'{c_memo}' and/or cc-memo '{cc_memo}' "
5378                     f"does not exist")
5379             raise NotExtracted()

```

Load the PDF. We only do this now so that we don't load it if there is nothing to extract.

```

5380         if not pdf:
5381             if not access_in(pdf_file):
5382                 warning(f"Cannot open '{pdf_file}'")
5383             endinput()
5384         try:

```

All safe, paranoia_in was already called above.

```

5385             pdf = pdfmw.PdfReader(pdf_file)
5386             except pdfmw.errors.PdfParseError as err:
5387                 error(rf"File '{pdf_file}' seems corrupted. Perhaps you "
5388                       rf"have to load Memoize earlier in the preamble",
5389                       f"In particular, Memoize must be loaded before "
5390                       f"TikZ library 'fadings' and any package "
5391                       f"deploying it, and in Beamer, load Memoize "
5392                       f"by writing \RequirePackage{{{memoize}}} before "
5393                       f"\documentclass{{{beamer}}}. "
5394                       f"This was the error thrown by Python: \n{err}")

```

Does the page exist?

```

5395         if page_n >= len(pdf.pages):
5396             error(rf"I cannot extract page {page_n} from '{pdf_file}', "
5397                 rf"as it contains only {len(pdf.pages)} page" +
5398                 ('s' if len(pdf.pages) > 1 else ''), '')

```

Check whether the page size matches the .mmz expectations.

```

5399         page = pdf.pages[page_n]
5400         expected_width_pt = float(m_ne['expected_width'])
5401         expected_height_pt = float(m_ne['expected_height'])
5402         mb = page['/MediaBox']
5403         width_bp = float(mb[2]) - float(mb[0])
5404         height_bp = float(mb[3]) - float(mb[1])
5405         width_pt = width_bp / 72 * 72.27
5406         height_pt = height_bp / 72 * 72.27
5407         if (abs(width_pt - expected_width_pt) > tolerance

```



```

5408         or abs(height_pt - expected_height_pt) > tolerance) \
5409         and not args.force:
5410     warning(
5411         f"I refuse to extract page {page_n+1} from '{pdf_file}' "
5412         f"because its size ({width_pt}pt x {height_pt}pt) "
5413         f"is not what I expected "
5414         f"({expected_width_pt}pt x {expected_height_pt}pt)")
5415     raise NotExtracted()

```

All tests were successful, let's create the extern file. First, the containing directory, if necessary.

```

5416         if dir_to_make:
5417             mkdir(dir_to_make)
5418             dir_to_make = None

```

Now the extern file. Note that `paranoia_out` was already called above.

```

5419         info(f"Page {page_n+1} --> {extern_file_out}")
5420         extern = pdfmw.PdfWriter(extern_file_out)
5421         extern.addpage(page)
5422         extern.write()

```

This page will get pruned.

```

5423         if args.prune:
5424             extern_pages.append(page_n)

```

Comment out this `\mmzNewExtern`.

```

5425         if not args.keep:
5426             line = '%' + line
5427     except NotExtracted:
5428         pass
5429     finally:
5430         if not args.keep:
5431             new_mmz.append(line)
5432     mmz.close()
5433     indent = ''
5434     texindent = ''
5435     info(done_message)

```

Write out the `.mmz` file with `\mmzNewExtern` lines commented out. (All safe, `paranoia_out` was already called above.)

```

5436         if not args.keep:
5437             with open(mmz_file, 'w') as mmz:
5438                 for line in new_mmz:
5439                     print(line, file = mmz, end = '')

```

Remove the extracted pages from the original PDF. (All safe, `paranoia_out` was already called above.)

```

5440         if args.prune and extern_pages:
5441             pruned_pdf = pdfmw.PdfWriter(pdf_file)
5442             pruned_pdf.addpages(
5443                 page for n, page in enumerate(pdf.pages) if n not in extern_pages)
5444             pruned_pdf.write()
5445             info(f"The following extern pages were pruned out of the PDF: " +
5446                 ", ".join(str(page+1) for page in extern_pages))

```

Report that extraction was successful.

```

5447         endinput()

```

Catch any errors in the script and output them to the log.

```

5448     except Exception as err:
5449         error(f'Python error: {err}', traceback.format_exc())

```

9.3 The Perl clean-up script `memoize-clean.pl`

```

5450 my $PROG = 'memoize-clean.pl';

```

```

5451 my $VERSION = '2024/03/15 v1.2.0';
5452
5453 use strict;
5454 use Getopt::Long;
5455 use Cwd 'realpath';
5456 use File::Spec;
5457 use File::Basename;
5458
5459 my $usage = "usage: $PROG [-h] [--yes] [--all] [--quiet] [--prefix PREFIX] " .
5460             "[mmz ...]\n";
5461 my $Help = <<END;
5462 Remove (stale) memo and extern files produced by package Memoize.
5463
5464 positional arguments:
5465     mmz                .mmz record files
5466
5467 options:
5468     -h, --help          show this help message and exit
5469     --version, -V       show version and exit
5470     --yes, -y           Do not ask for confirmation.
5471     --all, -a           Remove *all* memos and externs.
5472     --quiet, -q
5473     --prefix PREFIX, -p PREFIX
5474                        A path prefix to clean;
5475                        this option can be specified multiple times.
5476
5477 For details, see the man page or the Memoize documentation.
5478 END
5479
5480 my ($yes, $all, @prefixes, $quiet, $help, $print_version);
5481 GetOptions(
5482     "yes|y"    => \$yes,
5483     "all|a"    => \$all,
5484     "prefix|p=s" => \@prefixes,
5485     "quiet|q|?" => \$quiet,
5486     "help|h|?" => \$help,
5487     "version|V" => \$print_version,
5488     ) or die $usage;
5489 $help and die "$usage\n$Help";
5490 if ($print_version) { print("memoize-clean.pl of Memoize $VERSION\n"); exit 0 }
5491
5492 my (%keep, %prefixes);
5493
5494 my $curdir = Cwd::getcwd();
5495
5496 for my $prefix (@prefixes) {
5497     $prefixes{Cwd::realpath(File::Spec->catfile($curdir, $prefix))} = '';
5498 }
5499
5500 my @mmzs = @ARGV;
5501
5502 for my $mmz (@mmzs) {
5503     my ($mmz_filename, $mmz_dir) = File::Basename::fileparse($mmz);
5504     @ARGV = ($mmz);
5505     my $endinput = 0;
5506     my $empty = -1;
5507     my $prefix = "";
5508     while (<>) {
5509 if (/^ *$/) {
5510 } elsif ($endinput) {
5511     die "Bailing out, \\endinput is not the last line of file $mmz.\n";
5512 } elsif (/^ *\mmzPrefix *{(.*)}/) {
5513     $prefix = $1;

```

```

5514     $prefixes{Cwd::realpath(File::Spec->catfile(($curdir,$mmz_dir), $prefix))} = '';
5515     $empty = 1 if $empty == -1;
5516 } elsif (/^%? *\mmz(?:New|Used)(?:CC?Memo|Extern) *{(.*)}/) {
5517     my $fn = $1;
5518     if ($prefix eq '') {
5519 die "Bailing out, no prefix announced before file $fn.\n";
5520     }
5521     $keep{Cwd::realpath(File::Spec->catfile(($mmz_dir), $fn))} = 1;
5522     $empty = 0;
5523     if (rindex($fn, $prefix, 0) != 0) {
5524 die "Bailing out, prefix of file $fn does not match " .
5525     "the last announced prefix ($prefix).\n";
5526     }
5527 } elsif (/^ *\endinput *$/) {
5528     $endinput = 1;
5529 } else {
5530     die "Bailing out, file $mmz contains an unrecognized line: $_\n";
5531 }
5532 }
5533 die "Bailing out, file $mmz is empty.\n" if $empty && !$all;
5534 die "Bailing out, file $mmz does not end with \endinput; this could mean that " .
5535 "the compilation did not finish properly. You can only clean with --all.\n"
5536 if $endinput == 0 && !$all;
5537 }
5538
5539 my @tbdeleted;
5540 sub populate_tbdeleted {
5541     my ($basename_prefix, $dir, $suffix_dummy) = @_;
5542     opendir(MD, $dir) or die "Cannot open directory '$dir'";
5543     while( (my $fn = readdir(MD)) ) {
5544 my $path = File::Spec->catfile(($dir),$fn);
5545 if ($fn =~
5546     /\Q$basename_prefix\E[0-9A-F]{32}(?:-[0-9A-F]{32})?(?:-[0-9]+)?#
5547     (\.memo|(?:-[0-9]+)?\.pdf|\.log)/x
5548     and ($all || !exists($keep{$path}))) {
5549         push @tbdeleted, $path;
5550     }
5551     }
5552     closedir(MD);
5553 }
5554 for my $prefix (keys %prefixes) {
5555     my ($basename_prefix, $dir, $suffix);
5556     if (-d $prefix) {
5557 populate_tbdeleted('', $prefix, '');
5558     }
5559     populate_tbdeleted(File::Basename::fileparse($prefix));
5560 }
5561 @tbdeleted = sort(@tbdeleted);
5562
5563 my @allowed_dirs = ($curdir);
5564 my @deletion_not_allowed;
5565 for my $f (@tbdeleted) {
5566     my $f_allowed = 0;
5567     for my $dir (@allowed_dirs) {
5568 if ($f =~ /\Q$dir\E/) {
5569         $f_allowed = 1;
5570         last;
5571     }
5572     }
5573     push(@deletion_not_allowed, $f) if ! $f_allowed;
5574 }
5575 die "Bailing out, I was asked to delete these files outside the current directory:\n" .
5576     join("\n", @deletion_not_allowed) if (@deletion_not_allowed);

```

```

5577
5578 if (scalar(@tbdeleted) != 0) {
5579     my $a;
5580     unless ($yes) {
5581         print("I will delete the following files:\n" .
5582             join("\n",@tbdeleted) . "\n" .
5583             "Proceed (y/n)? ");
5584         $a = lc(<>);
5585         chomp $a;
5586     }
5587     if ($yes || $a eq 'y' || $a eq 'yes') {
5588         foreach my $fn (@tbdeleted) {
5589             print "Deleting ", $fn, "\n" unless $quiet;
5590             unlink $fn;
5591         }
5592     } else {
5593         die "Bailing out.\n";
5594     }
5595 } elsif (!$quiet) {
5596     print "Nothing to do, the directory seems clean.\n";
5597 }

```

9.4 The Python clean-up script memoize-clean.py

```

5598 __version__ = '2024/03/15 v1.2.0'
5599
5600 import argparse, re, sys, pathlib, os
5601
5602 parser = argparse.ArgumentParser(
5603     description="Remove (stale) memo and extern files.",
5604     epilog = "For details, see the man page or the Memoize documentation "
5605             "(https://ctan.org/pkg/memoize).")
5606 )
5607 parser.add_argument('--yes', '-y', action = 'store_true',
5608                     help = 'Do not ask for confirmation.')
5609 parser.add_argument('--all', '-a', action = 'store_true',
5610                     help = 'Remove *all* memos and externs.')
5611 parser.add_argument('--quiet', '-q', action = 'store_true')
5612 parser.add_argument('--prefix', '-p', action = 'append', default = [],
5613                     help = 'A path prefix to clean; this option can be specified multiple times.')
5614 parser.add_argument('mmz', nargs= '*', help='.mmz record files')
5615 parser.add_argument('--version', '-V', action = 'version',
5616                     version = f"%(prog)s of Memoize " + __version__)
5617 args = parser.parse_args()
5618
5619 re_prefix = re.compile(r'\\mmzPrefix *{(.*)}')
5620 re_memo = re.compile(r'%? *\\mmz(?:New|Used)(?:CC?Memo|Extern) *{(.*)}')
5621 re_endinput = re.compile(r' *\\endinput *$')
5622
5623 prefixes = set(pathlib.Path(prefix).resolve() for prefix in args.prefix)
5624 keep = set()

```

We loop through the given .mmz files, adding prefixes to whatever manually specified by the user, and collecting the files to keep.

```

5625 for mmz_fn in args.mmz:
5626     mmz = pathlib.Path(mmoz_fn)
5627     mmz_parent = mmz.parent.resolve()
5628     try:
5629         with open(mmz) as mmz_fh:
5630             prefix = ''
5631             endinput = False
5632             empty = None
5633             for line in mmz_fh:

```

```

5634         line = line.strip()
5635
5636     if not line:
5637         pass
5638
5639     elif endinput:
5640         raise RuntimeError(
5641             rf'Bailing out, '
5642             rf'\endinput is not the last line of file {mmz_fn}.' )
5643
5644     elif m := re_prefix.match(line):
5645         prefix = m[1]
5646         prefixes.add( (mmz_parent/prefix).resolve() )
5647         if empty is None:
5648             empty = True
5649
5650     elif m := re_memo.match(line):
5651         if not prefix:
5652             raise RuntimeError(
5653                 f'Bailing out, no prefix announced before file "{m[1]}".')
5654         if not m[1].startswith(prefix):
5655             raise RuntimeError(
5656                 f'Bailing out, prefix of file "{m[1]}" does not match '
5657                 f'the last announced prefix ({prefix}).')
5658         keep.add((mmz_parent / m[1]))
5659         empty = False
5660
5661     elif re_endinput.match(line):
5662         endinput = True
5663         continue
5664
5665     else:
5666         raise RuntimeError(fr"Bailing out, "
5667                             fr"file {mmz_fn} contains an unrecognized line: {line}")
5668
5669     if empty and not args.all:
5670         raise RuntimeError(fr'Bailing out, file {mmz_fn} is empty.')
5671
5672     if not endinput and empty is not None and not args.all:
5673         raise RuntimeError(
5674             fr'Bailing out, file {mmz_fn} does not end with \endinput; '
5675             fr'this could mean that the compilation did not finish properly. '
5676             fr'You can only clean with --all.'
5677         )

```

It is not an error if the file doesn't exist. Otherwise, cleaning from scripts would be cumbersome.

```

5678     except FileNotFoundError:
5679         pass
5680
5681     tbdeleted = []
5682     def populate_tbdeleted(folder, basename_prefix):
5683         re_aux = re.compile(
5684             re.escape(basename_prefix) +
5685             '[0-9A-F]{32}(?:-[0-9A-F]{32})?'
5686             '(?:-[0-9]+)?(?:\.memo|(?:-[0-9]+)?\.pdf|\.log)$')
5687         try:
5688             for f in folder.iterdir():
5689                 if re_aux.match(f.name) and (args.all or f not in keep):
5690                     tbdeleted.append(f)
5691         except FileNotFoundError:
5692             pass
5693
5694     for prefix in prefixes:

```

```

    "prefix" is interpreted both as a directory (if it exists) and a basename prefix.
5695     if prefix.is_dir():
5696         populate_tbdeleted(prefix, '')
5697     populate_tbdeleted(prefix.parent, prefix.name)
5698
5699 allowed_dirs = [pathlib.Path().absolute()] # todo: output directory
5700 deletion_not_allowed = [f for f in tbdeleted if not f.is_relative_to(*allowed_dirs)]
5701 if deletion_not_allowed:
5702     raise RuntimeError("Bailing out, "
5703         "I was asked to delete these files outside the current directory:\n" +
5704         "\n".join(str(f) for f in deletion_not_allowed))
5705
5706 _cwd_absolute = pathlib.Path().absolute()
5707 def relativize(path):
5708     try:
5709         return path.relative_to(_cwd_absolute)
5710     except ValueError:
5711         return path
5712
5713 if tbdeleted:
5714     tbdeleted.sort()
5715     if not args.yes:
5716         print('I will delete the following files:')
5717         for f in tbdeleted:
5718             print(relativize(f))
5719         print("Proceed (y/n)? ")
5720         a = input()
5721     if args.yes or a == 'y' or a == 'yes':
5722         for f in tbdeleted:
5723             if not args.quiet:
5724                 print("Deleting", relativize(f))
5725             try:
5726                 f.unlink()
5727             except FileNotFoundError:
5728                 print(f"Cannot delete {f}")
5729     else:
5730         print("Bailing out.")
5731 elif not args.quiet:
5732     print('Nothing to do, the directory seems clean.')

```

Index

Numbers written in red refer to the code line where the corresponding entry is defined; numbers in blue refer to the code lines where the entry is used.

Symbols

| | | | |
|--------------------------------------------|------|--------------------------------------|------|
| .install advice (/handlers) | 2208 | force reframe | 1753 |
| /handlers/.meaning to context (/mmz) | 827 | inner handler | 2267 |
| /handlers/.value to context (/mmz) | 827 | integrated driver | 694 |
| /advice/install keys: | | memoize | 1678 |
| activation | 2220 | multiref | 1766 |
| setup key | 2220 | noop | 1686 |
| /advice keys: | | options | 2267 |
| activation/deferred | 2369 | outer handler | 2267 |
| activation/immediate | 2369 | raw collector options | 2267 |
| /collargs keys: | | ref | 1740 |
| alias | 3009 | refrange | 1753 |
| append expandable postprocessor | 2964 | replicate | 1700 |
| append expandable preprocessor | 2964 | reset | 2291 |
| append postprocessor | 2946 | run conditions | 2267 |
| append preprocessor | 2946 | run if memoization is possible | 1649 |
| begin tag | 2930 | run if memoizing | 1657 |
| braces | 2922 | to context | 1715 |
| caller | 2856 | volcite | 1914 |
| clear args | 2994 | volcites | 1914 |
| clear postprocessors | 2956 | /mmz keys: | |
| clear preprocessors | 2956 | /handlers/.meaning to context | 827 |
| end tag | 2930 | /handlers/.value to context | 827 |
| environment | 2925 | activate | 2380 |
| fix from no verbatim | 2903 | activate csname | 2255 |
| fix from verb | 2903 | activate deferred | 2254 |
| fix from verbatim | 2903 | activate key | 2257 |
| ignore nesting | 2938 | activation | 2252 |
| ignore other tags | 2942 | after memoization | 641 |
| no delimiters | 2990 | at begin memoization | 641 |
| no verbatim | 2893 | at end memoization | 641 |
| prepend expandable postprocessor | 2964 | auto | 2229 |
| prepend expandable preprocessor | 2964 | auto csname | 2229 |
| prepend postprocessor | 2946 | auto csname' | 2229 |
| prepend preprocessor | 2946 | auto key | 2229 |
| return | 2999 | auto key' | 2229 |
| tags | 2930 | auto' | 2229 |
| verb | 2893 | bat | 1370 |
| verbatim | 2893 | begindocument | 2105 |
| verbatim ranges | 4024 | begindocument/before | 2105 |
| /handlers keys: | | begindocument/end | 2105 |
| .install advice | 2208 | capture | 682 |
| /mmz/auto keys: | | clear context | 809 |
| after setup | 2292 | context | 809 |
| apply options | 1665 | csname meaning to context | 827 |
| args | 2267 | deactivate | 2380 |
| bailout handler | 2267 | deactivate csname | 2255 |
| biblatex ccmemo cite | 1942 | deactivate key | 2257 |
| cite | 1914 | direct ccmemo input | 927 |
| cites | 1914 | disable | 238 |
| clear collector options | 2267 | driver | 670 |
| clear options | 2267 | enable | 238 |
| clear raw collector options | 2267 | enddocument/afterlastpage | 2105 |
| collector | 2267 | extract | 1237 |
| collector options | 2267 | extract/perl | 1242 |
| force multiref | 1766 | extract/python | 1242 |
| force ref | 1740 | extract/tex | 1426 |
| | | force activate | 2380 |

| | | | |
|---------------------------|------|---------------------------------------------|------------------------------------------------------------------------------------------------|
| ignore spaces | 339 | \advice@handle@rc | 2487, 2489, 2489, 2577 |
| include context in ccmemo | 923 | \advice@if@our@definition | 2434, 2453, 2465, 2465 |
| include source in cmemo | 870 | \advice@init@I | 2345, 2346, 2348, 2504, 2602, 2646, 2650 |
| key meaning to context | 827 | \advice@init@i | 2345, 2345, 2347, 2484, 2586, 2645, 2649 |
| key value to context | 827 | \advice@original@cs | 2397, 2398, 2494, 2581, 2592, 2662 |
| makefile | 1398 | \advice@original@csname | 2397, 2397, 2433, 2452, 2474, 2479, 2480, 2562, 2565, 2570, 2571, 2578 |
| manual | 1601 | \advice@pgfkeys@collector | 2245, 2250, 2541, 2541 |
| meaning to context | 827 | \advice@setup@save | 2320, 2349, 2349 |
| memo dir | 304 | \advice@trace | 2652, 2658, 2660, 2661, 2666, 2674, 2683, 2689, 2692, 2695, 2698, 2708, 2711, 2714 |
| mkdir | 287 | \advice@trace@init@I | 2646, 2659, 2680 |
| mkdir command | 287 | \advice@trace@init@i | 2645, 2659, 2659 |
| no memo dir | 304 | \advice@typeout | 715, 2652, 2653, 2655, 2656, 2658 |
| no record | 1315 | \AdviceArgs | 2275, 2334, 2342, 2362, 2503, 2519, 2529, 2535, 2694 |
| no verbatim | 343 | \AdviceBailoutHandler | 2268, 2324, 2354, 2482, 2493, 2591, 2673, 2676, 2677 |
| normal | 252 | \AdviceCollector | 1668, 1965, 2004, 2019, 2270, 2284, 2325, 2326, 2358, 2503, 2688, 2691, 2705 |
| options | 250 | \AdviceCollectorOptions | 2271, 2272, 2327, 2361, 2503, 2525, 2526, 2697 |
| padding | 1024 | \AdviceCsnameGetOriginal | 1721, 2406 |
| padding bottom | 1019 | \AdviceGetOriginal | 2397, 2503, 2601 |
| padding left | 1019 | \AdviceIfArgs | 2517, 2533 |
| padding right | 1019 | \AdviceInnerHandler | 2276, 2328, 2363, 2503, 2530, 2542, 2710, 2717, 2719, 2777 |
| padding to context | 1029 | \AdviceName | 1693, 1996, 2296, 2339, 2497, 2500, 2502, 2513, 2518, 2523, 2598, 2642 |
| padding top | 1019 | \AdviceNamespace | 1718, 1721, 2497, 2499, 2597, 2641 |
| prefix | 261 | \AdviceOptions | 1674, 1703, 2277, 2278, 2280, 2329, 2364, 2503, 2716 |
| readonly | 252 | \AdviceOriginal | 1682, 1687, 1697, 1709, 1726, 1743, 1764, 1776, 1782, 1795, 1953, 1976, 2027, 2497, 2503, 2601 |
| recompile | 252 | \AdviceOuterHandler | 2269, 2325, 2357, 2503, 2505, 2603, 2682, 2685, 2686 |
| record | 1299 | \AdviceRawCollectorOptions | 1676, 2273, 2274, 2333, 2338, 2359, 2360, 2503, 2524, 2700, 2703 |
| record/bat/... | 1386 | \AdviceReplaced | 1681, 1708, 2497, 2502, 2600 |
| record/makefile/... | 1405 | \AdviceRunConditions | 2267, 2323, 2353, 2482, 2486, 2587, 2664, 2665, 2668 |
| record/mmz/... | 1337 | \AdviceSetup | 2230, 2234, 2296 |
| record/sh/... | 1374 | \AdviceTracingOff | 2644 |
| sh | 1370 | \AdviceTracingOn | 2644 |
| tex extraction command | 1461 | \AdviceType | 1625, 2291, 2296 |
| tex extraction options | 1461 | after memoization (/mmz) | 641 |
| tex extraction script | 1461 | after setup (/mmz/auto) | 2292 |
| tracing | 760 | alias (/collargs) | 3009 |
| try activate | 2380 | append expandable postprocessor (/collargs) | 2964 |
| verb | 343 | append expandable preprocessor (/collargs) | 2964 |
| verbatim | 343 | append postprocessor (/collargs) | 2946 |
| | | append preprocessor (/collargs) | 2946 |
| | | apply options (/mmz/auto) | 1665 |
| | | args (/mmz/auto) | 2267 |
| | | at begin memoization (/mmz) | 641 |
| | | at end memoization (/mmz) | 641 |
| | | auto (/mmz) | 2229 |
| | | auto csname (/mmz) | 2229 |
| | | auto csname' (/mmz) | 2229 |
| | | auto key (/mmz) | 2229 |
| | | auto key' (/mmz) | 2229 |
| | | auto' (/mmz) | 2229 |

A

| | |
|--------------------------------|------------------------|
| abortOnError (Lua function) | 562 |
| activate (/mmz) | 2380 |
| activate csname (/mmz) | 2255 |
| activate deferred (/mmz) | 2254 |
| activate key (/mmz) | 2257 |
| activation (/advice/install) | 2220 |
| activation (/mmz) | 2252 |
| activation/deferred (/advice) | 2369 |
| activation/immediate (/advice) | 2369 |
| \advice@activate | 2382, 2419, 2419 |
| \advice@activate@cmd | 2421, 2431, 2431, 2547 |
| \advice@activate@cmd@do | 2438, 2444, 2473, 2473 |
| \advice@activate@env | 2423, 2544, 2545, 2561 |
| \advice@begin@env@outer | 2589, 2596, 2596 |
| \advice@begin@env@rc | 2579, 2585, 2585 |
| \advice@begin@rc | 2293, 2577, 2577 |
| \advice@CollectArgumentsRaw | 2285, 2326, 2516, 2516 |
| \advice@deactivate | 2387, 2419, 2425 |
| \advice@deactivate@cmd | 2427, 2450, 2450, 2554 |
| \advice@deactivate@cmd@do | 2454, 2478, 2478 |
| \advice@deactivate@env | 2429, 2544, 2552, 2569 |
| \advice@handle | 2466, 2475, 2483, 2483 |
| \advice@handle@outer | 2491, 2498, 2498 |

| | | |
|-----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|--|
| B | | |
| bailout handler (/mmz/auto) | 2267 | |
| bat (/mmz) | 1370 | |
| begin tag (/collargs) | 2930 | |
| begindocument (/mmz) | 2105 | |
| begindocument/before (/mmz) | 2105 | |
| begindocument/end (/mmz) | 2105 | |
| biblatex ccmemo cite (/mmz/auto) | 1942 | |
| braces (/collargs) | 2922 | |
| C | | |
| caller (/collargs) | 2856 | |
| capture (/mmz) | 682 | |
| \catcodetable@atletter | 3989, 3992 | |
| \cite | 2030 | |
| cite (/mmz/auto) | 1914 | |
| \cites | 2041 | |
| cites (/mmz/auto) | 1914 | |
| clear args (/collargs) | 2994 | |
| clear collector options (/mmz/auto) | 2267 | |
| clear context (/mmz) | 809 | |
| clear options (/mmz/auto) | 2267 | |
| clear postprocessors (/collargs) | 2956 | |
| clear preprocessors (/collargs) | 2956 | |
| clear raw collector options (/mmz/auto) | 2267 | |
| \collargs@ | 2836, 3015, 3293, 3301, 3323, 3343, 3364, 3370, 3415, 3422, 3426, 3466, 3472, 3524, 3544, 3559, 3565, 3588, 3596, 3712, 3900, 3913 | |
| \collargs&& | 3049 | |
| \collargs@! | 3074 | |
| \collargs@+ | 3068 | |
| \collargs@. | 3080 | |
| \collargs@@ | 3013, 3017, 3019, 3062, 3066, 3073 | |
| \collargs@appendarg | 3095, 3095, 3292, 3299, 3322, 3342, 3369, 3408, 3412, 3425, 3465, 3471, 3523, 3542, 3558, 3586, 3594, 3711, 3898, 3899 | |
| \collargs@b | 3601, 3601 | |
| \collargs@bgroups | 3942, 3951, 3963, 3963, 4059, 4102 | |
| \collargs@catcodetable@initex | 3989, 4001, 4003, 4009 | |
| \collargs@catcodetable@verbatim | 3989, 3991, 3996, 3997, 4000, 4002, 4020, 4055 | |
| \collargs@ccc | 3184, 3187, 3507, 3515, 3532, 3965, 3966, 3971 | |
| \collargs@ccs@cases | 838, 2300, 2420, 2426, 2839, 2839, 2860 | |
| \collargs@D | 3383, 3383 | |
| \collargs@d | 3350, 3350, 3383, 3384, 3385 | |
| \collargs@defcollector | 3120, 3120, 3339, 3366, 3592 | |
| \collargs@defusecollector | 3120, 3124, 3291, 3297, 3320, 3340, 3367, 3463, 3470 | |
| \collargs@E | 3914, 3914 | |
| \collargs@e | 3849, 3849, 3914 | |
| \collargs@egroups | 3943, 3953, 3963, 3964, 4061, 4113 | |
| \collargs@fix | 3029, 3084, 4139, 4139, 4423 | |
| \collargs@fix@NtoN | 4160, 4160–4162 | |
| \collargs@fix@NtoV | 4173, 4178, 4178, 4227, 4230 | |
| \collargs@fix@Ntov | 4163, 4163 | |
| \collargs@fix@VtoN | 4218, 4238, 4249, 4249 | |
| \collargs@fix@vtov | 4208, 4208 | |
| \collargs@fix@VtoN@comment | 4254, 4264, 4283 | |
| \collargs@fix@VtoN@escape | 4251, 4287, 4287 | |
| \collargs@fix@VtoN@token | 4256, 4261, 4261 | |
| \collargs@fix@VtoV | 4160, 4162 | |
| \collargs@fix@Vtov | 4236 | |
| \collargs@fix@vtov | 4222, 4222 | |
| \collargs@fix@vtov | 4160, 4161 | |
| \collargs@forrange | 3256, 3256, 3274, 4018, 4019 | |
| \collargs@forranges | 3272, 3272, 4074, 4130 | |
| \collargs@G | 3570, 3570 | |
| \collargs@g | 3561, 3561, 3570 | |
| \collargs@grabbed@spaces | 3106, 3220, 3220, 3233, 3412, 3784, 3810 | |
| \collargs@grabspaces | 3140, 3140, 3215, 3347, 3406, 3449, 3476, 3575, 3694, 3774, 3805, 3863 | |
| \collargs@ifnextcat | 3236, 3236, 3602 | |
| \collargs@init@grabspaces | 2831, 3107, 3135, 3135, 3230, 3411, 3785, 3811 | |
| \collargs@insert@char | 4262, 4439, 4440, 4451, 4461, 4486 | |
| \collargs@l | 3279, 3279 | |
| \collargs@letusecollector | 3120, 3129, 3747, 3775, 3806, 3817, 3830, 3845 | |
| \collargs@m | 3441, 3441, 3568, 3901 | |
| \collargs@make@char | 4106, 4117, 4439, 4445, 4455, 4465, 4495 | |
| \collargs@make@no@verbatim | 3932, 4123, 4124, 4128 | |
| \collargs@make@verbatim | 3919, 3925, 4029, 4030, 4065 | |
| \collargs@make@verbatim@bgroup | 4082, 4098, 4098 | |
| \collargs@make@verbatim@comment | 4088, 4120, 4120 | |
| \collargs@make@verbatim@egroup | 4085, 4109, 4109 | |
| \collargs@maybegrabspaces | 3213, 3213, 3381, 3435 | |
| \collargs@O | 3385, 3385 | |
| \collargs@o | 3384, 3384 | |
| \collargs@other@bgroup | 3298, 3302, 3490, 3548, 3556, 3578, 3579, 3584, 3638, 3649, 3979, 3984, 4106 | |
| \collargs@other@egroup | 3493, 3549, 3556, 3580, 3584, 3638, 3649, 3979, 3985, 4117 | |
| \collargs@percentchar | 879, 881, 937, 949, 3275, 3277, 4034 | |
| \collargs@R | 3349, 3349 | |
| \collargs@r | 3328, 3328, 3349 | |
| \collargs@readContent | 3552, 3590, 3626, 3733, 3733 | |
| \collargs@reinsert@spaces | 3227, 3227, 3280, 3317, 3626 | |
| \collargs@s | 3440, 3440 | |
| \collargs@t | 3386, 3386, 3440 | |
| \collargs@u | 3306, 3306 | |
| \collargs@v | 3571, 3571 | |
| \collargs@verbatim@ranges | 4024, 4025, 4027, 4028, 4033, 4074, 4130 | |
| \collargs@verbatim@wrap | 2830, 2893, 2898, 3061, 3065 | |
| \collargs@wrap | 3110, 3110, 3321, 3341, 3368, 3464, 3557, 3585, 3593, 3731, 3904 | |
| \collargsAlias | 1968, 3009 | |
| \collargsAppendExpandablePostprocessor | 324, 438, 1909, 2964 | |
| \collargsAppendExpandablePreprocessor | 2964 | |
| \collargsAppendPostprocessor | 2946 | |
| \collargsAppendPreprocessor | 1971, 2946 | |
| \collargsArg | 324, 1909, 1971, 2972, 2977, 2982, 2987, 3079, 3096, 3106, 3113, 3115, 3527, 3533, 3542, 3558, 3586, 3711, 3745, 3790, 3822, 3839, 3899, 3909, 4432 | |
| \collargsArgs | 1976, 1996, 2002, 2027, 2736, 2834, 3078, 3087, 3089, 3106 | |
| \collargsBraces | 2923, 3573, 3979 | |
| \collargsCaller | 387, 434, 2523, 2856 | |
| \collargsClearPostprocessors | 2956 | |
| \collargsClearPreprocessors | 2956 | |
| \collargsEnvironment | 2339, 2925 | |
| \collargsFixFromNoVerbatim | 366, 399, 419, 2903 | |
| \collargsFixFromVerb | 2903 | |

| | | | |
|-------------------------------------------------------|------------------------------------|--------------------------------------------------------|--------------------------|
| \collargsFixFromVerbatim | 2903 | \ifcollargsAddTags | 2930 |
| \collargsNoVerbatim | 355, 2896, 3915 | \ifcollargsBeginTag | 2930, 3717 |
| \collargsPrependExpandablePostprocessor | 2964 | \ifcollargsClearArgs | 2833, 2994 |
| \collargsPrependExpandablePreprocessor | 2964 | \ifcollargsEndTag | 2930, 3724 |
| \collargsPrependPostprocessor | 2946 | \ifcollargsIgnoreNesting | 2938, 3821 |
| \collargsPrependPreprocessor | 2946 | \ifcollargsIgnoreOtherTags | 2942, 3616, 3686 |
| \collargsReturn | 2722, 2999, 3086 | \ifcollargsNoDelimiters | 2990, 3101 |
| \collargsSet | 2526, 2824, | \ifinmemoize | 489, 559, 1651 |
| 2838, 2856, 2893, 2903, 2922, 2925, 2930, 2938, 2942, | | \ifmemoize | 238, 523, 721, 1650 |
| 2946, 2958, 2964, 2990, 2994, 2999, 3009, 3065, 4024 | | \IfMemoizing | 703, 2171 |
| \collargsVerb | 351, 1908, 2895, 3915 | \ifmemoizing | |
| \collargsVerbatim | 347, 2894, 3574, 3915 | .. 487, 521, 646, 654, 662, 811, 819, 1658, 1856, 1956 | |
| \collargsVerbatimRanges | 4024 | \ifmmz@abort | 535, 617, 638, 638, 1000 |
| \CollectArguments | 2821, 2890 | \ifmmz@direct@ccmemo@input | |
| \CollectArgumentsRaw | 321, 386, 433, 2522, 2821 | 751, 927, 927, 953, 970, 980, 990 | |
| collector (/mmz/auto) | 2267 | \ifmmz@include@context | 923, 923, 938 |
| collector options (/mmz/auto) | 2267 | \ifmmz@include@source | 870, 873, 882 |
| context (/mmz) | 809 | \ifmmzkeepexterns | 674, 1071 |
| csname meaning to context (/mmz) | 827 | \ifmmzUnmemoizable | 532, 610, 640 |
| D | | ignore nesting (/collargs) | 2938 |
| deactivate (/mmz) | 2380 | ignore other tags (/collargs) | 2942 |
| deactivate csname (/mmz) | 2255 | ignore spaces (/mmz) | 339 |
| deactivate key (/mmz) | 2257 | include context in ccmemo (/mmz) | 923 |
| direct ccmemo input (/mmz) | 927 | include source in cmemo (/mmz) | 870 |
| disable (/mmz) | 238 | inner handler (/mmz/auto) | 2267 |
| driver (/mmz) | 670 | integrated driver (/mmz/auto) | 694 |
| E | | K | |
| enable (/mmz) | 238 | key meaning to context (/mmz) | 827 |
| end tag (/collargs) | 2930 | key value to context (/mmz) | 827 |
| enddocument/afterlastpage (/mmz) | 2105 | L | |
| environment (/collargs) | 2925 | Lua functions: | |
| environments: | | abortOnError | 562 |
| memoize | 393 | M | |
| nomemoize | 451 | makefile (/mmz) | 1398 |
| \etoksapp | 2802, 3762, 3784, 3790, 3810, 3839 | manual (/mmz) | 1601 |
| extract (/mmz) | 1237 | meaning to context (/mmz) | 827 |
| extract/perl (/mmz) | 1242 | memo dir (/mmz) | 304 |
| extract/python (/mmz) | 1242 | \Memoize | 382, 430, 506, 720, 1680 |
| extract/tex (/mmz) | 1426 | \memoize | 408, 481 |
| F | | memoize (/mmz/auto) | 1678 |
| \filetotoks | 214, 993 | memoize (env.) | 393 |
| fix from no verbatim (/collargs) | 2903 | \memoizinggrouplevel | 593, 636, 705 |
| fix from verb (/collargs) | 2903 | mkdir (/mmz) | 287 |
| fix from verbatim (/collargs) | 2903 | mkdir command (/mmz) | 287 |
| force activate (/mmz) | 2380 | \mmz | 359, 450 |
| force multiref (/mmz/auto) | 1766 | \mmz@auto@bailout | 1616, 1628, 1665, 1670 |
| force ref (/mmz/auto) | 1740 | \mmz@auto@label | 1737, 1778, 1778 |
| force refrange (/mmz/auto) | 1753 | \mmz@auto@memoize | 1622, 1678, 1678 |
| G | | \mmz@auto@multiref | 1766, 1768, 1770, 1773 |
| \gtoksapp | 612, 812, 1707, 1866, 1884, 2802 | \mmz@auto@noop | 1626, 1630, 1686, 1686 |
| I | | \mmz@auto@noop@env | 1628, 1686, 1692 |
| \ifAdviceRun | 2490, 2515, 2588, 2670 | \mmz@auto@outer | 1617, 1665, 1665 |
| \ifcollargs@fix@requested | 4138, 4138, 4141 | \mmz@auto@rc@if@memoization@possible | 1612, 1649, 1649 |
| \ifcollargs@verbatim | 2827, | \mmz@auto@rc@if@memoizing | 1614, 1657, 1657 |
| 2891, 2891, 3033, 3164, 3221, 3307, 3329, 3351, | | \mmz@auto@ref | 1731, 1732, 1740, 1740 |
| 3387, 3442, 3609, 3631, 3642, 3657, 3668, 3852, 4149 | | \mmz@auto@refrange | 1753, 1755, 1757, 1760 |
| \ifcollargs@verbatimbraces | | \mmz@auto@replicate | 1632, 1700, 1700 |
| 2828, 2891, 2892, 3034, 3284, 3479, 3614, | | \mmz@auto@tocontext | 1633, 1715, 1715 |
| 3637, 3648, 3683, 3720, 3727, 4056, 4099, 4110, 4150 | | \mmz@biblatex@cite@inner | 1919, 1926, 1973, 1973 |
| | | \mmz@biblatex@cite@outer | 1916, 1923, 1963, 1963 |

| | | | |
|----------------------------------|--------------------------------------------------------|----------------------------------------------|-----------------|
| \mmz@biblatex@cite@rc | 1915, 1929, 1955, 1955 | \mmzTracingOff | 760, 2181 |
| \mmz@biblatex@cites@outer | 1930, 1937, 2001, 2001 | \mmzTracingOn | 760, 2180 |
| \mmz@biblatex@entry | 1905, 1950, 1950 | \mmzUnmemoizable | 639, 1639, 2170 |
| \mmz@biblatex@mark@citation@key | 1967, 1968, 1970 | \mmzUsedCCMemo | 1364, 1433 |
| \mmz@ccmemo@append@resource | 946, 946, 1056 | \mmzUsedCMemo | 1361, 1433 |
| \mmz@ccmemo@resources | 598, 934, 1055, 1062, 1062 | \mmzUsedExtern | 1358, 1433 |
| \mmz@compile | 520, 553, 553, 1003 | multiref (/mmz/auto) | 1766 |
| \mmz@compute@context@mdfivesum | 620, 623, 902, 904, 904 | | |
| \mmz@if@roughly@equal | 1183, 1184, 1527, 1530, 1557, 1558 | N | |
| \mmz@include@extern | 967, 1165, 1165 | no delimiters (/collargs) | 2990 |
| \mmz@include@extern@from@tbe@box | 632, 1220, 1220 | no memo dir (/mmz) | 304 |
| \mmz@insertpdfpage | 1172, 1201, 1202, 1209, 1213 | no record (/mmz) | 1315 |
| \mmz@inverse@mag | | no verbatim (/collargs) | 2893 |
| | 1132, 1137, 1139, 1141, 1143–1146, 1148, 1150 | no verbatim (/mmz) | 343 |
| \mmz@lua@atbeginmemoization | 562, 574, 586, 604 | \nomemoize | 460, 481 |
| \mmz@lua@atendmemoization | 562, 577, 587, 609 | nomemoize (env.) | 451 |
| \mmz@maybe@scantokens | 490, 492, 517, 1687 | \nommmz | 441 |
| \mmz@memoize | 528, 540, 589, 589, 1005 | \nommmzkeys | 233 |
| \mmz@process@ccmemo | 545, 961, 961 | noop (/mmz/auto) | 1686 |
| \mmz@process@cmemo | 531, 890, 890 | normal (/mmz) | 252 |
| \mmz@remove@quotes | 298, 311, 311 | | |
| \mmz@shipout@extern | 1074, 1078, 1078 | O | |
| \mmz@shipout@externs | 626, 1064, 1064 | options (/mmz/auto) | 2267 |
| \mmz@shipout@mag | 1086, 1130, 1130 | options (/mmz) | 250 |
| \mmz@split@prefix | 264 | outer handler (/mmz/auto) | 2267 |
| \mmz@tbe@box | 597, 1054, 1063, 1063, 1071, 1224 | | |
| \mmz@write@ccmemo | 625, 931, 931 | P | |
| \mmz@write@cmemo | 613, 621, 624, 875, 875 | padding (/mmz) | 1024 |
| \mmzAbort | 637, 1636, 1750, 1985, 2169 | padding bottom (/mmz) | 1019 |
| \mmzAfterMemoization | 633, 641 | padding left (/mmz) | 1019 |
| \mmzAtBeginMemoization | 605, 641, 1883 | padding right (/mmz) | 1019 |
| \mmzAtEndMemoization | 607, 641 | padding to context (/mmz) | 1029 |
| \mmzAutoInit | 1667, 1673, 1694 | padding top (/mmz) | 1019 |
| \mmzCCMemo | 600, 676, 679, 922, | prefix (/mmz) | 261 |
| | 937, 1707, 1779, 1823, 1866, 1870, 1988, 1993, 2177 | prepend expandable postprocessor (/collargs) | 2964 |
| \mmzCMemo | 599, 612, 869, 881, 1864, 1884, 2176 | prepend expandable preprocessor (/collargs) | 2964 |
| \mmzContext | 807, 814, 822, 905, 2174 | prepend postprocessor (/collargs) | 2946 |
| \mmzContextExtra | | prepend preprocessor (/collargs) | 2946 |
| | 601, 611, 807, 812, 820, 879, 895, 905, 2175 | | |
| \mmzEndMemo | 942, 956, 961 | R | |
| \mmzExternalizeBox | 678, 1037 | raw collector options (/mmz/auto) | 2267 |
| \mmzExternPages | 1120, 1125, 1155, 1158–1160, 2178 | readonly (/mmz) | 252 |
| \mmzExtraPages | 1125, 1164, 1792, 1793, 1795, 2179 | recompile (/mmz) | 252 |
| \mmzForceNoRef | 1732, 1745, 1757, 1770, 2173 | record (/mmz) | 1299 |
| \mmzLabel | 1780, 1784 | record/bat/... (/mmz) | 1386 |
| \mmzNewCCMemo | 1355, 1433 | record/makefile/... (/mmz) | 1405 |
| \mmzNewCMemo | 1352, 1433 | record/mmz/... (/mmz) | 1337 |
| \mmzNewExtern | 1347, 1439 | record/sh/... (/mmz) | 1374 |
| \mmznext | 1660, 1671, 1675 | \ref | 1735 |
| \mmzNoRef | 1731, 1745, 1755, 1768, 2172 | ref (/mmz/auto) | 1740 |
| \mmzPrefix | 1343, 1433 | refrange (/mmz/auto) | 1753 |
| \mmzRegularPages | 129, 1123, 1152 | replicate (/mmz/auto) | 1700 |
| \mmzResource | 949, 951 | reset (/mmz/auto) | 2291 |
| \mmzset | 228, 242, 256, 261, 281, 287, 304, | return (/collargs) | 2999 |
| | 340, 345, 371, 402, 424, 644, 670, 682, 694, 760, 809, | run conditions (/mmz/auto) | 2267 |
| | 870, 924, 928, 1019, 1127, 1149, 1156, 1238, 1299, | run if memoization is possible (/mmz/auto) | 1649 |
| | 1315, 1329, 1331, 1337, 1405, 1426, 1466, 1597, 1602, | run if memoizing (/mmz/auto) | 1657 |
| | 1635, 1638, 1642, 1644, 1674, 1675, 1729, 1734, 1753, | | |
| | 1766, 1794, 1798, 1831, 1838, 1845, 1853, 1890, 1894, | S | |
| | 1903, 2029, 2106–2109, 2112, 2117, 2124, 2134, 2141 | setup key (/advice/install) | 2220 |
| \mmzSingleExternDriver | 672, 675 | sh (/mmz) | 1370 |
| \mmzSource | 883, 889 | | |
| \mmzThisContext | 939, 961 | T | |
| | | tags (/collargs) | 2930 |

| | | | |
|-----------------------------------------------------|------|-----------------------------------|------|
| tex extraction command (/mmz) | 1461 | verbatim (/collargs) | 2893 |
| tex extraction options (/mmz) | 1461 | verbatim (/mmz) | 343 |
| tex extraction script (/mmz) | 1461 | verbatim ranges (/collargs) | 4024 |
| to context (/mmz/auto) | 1715 | \volcite | 2065 |
| \toksapp | 220, | volcite (/mmz/auto) | 1914 |
| 814, 2764, 2768, 2774, 2775, 2802, 3533, 3741, 3816 | | \volcites | 2067 |
| tracing (/mmz) | 760 | volcites (/mmz/auto) | 1914 |
| try activate (/mmz) | 2380 | | |

V

| | |
|------------------------|------|
| verb (/collargs) | 2893 |
| verb (/mmz) | 343 |

X

| | |
|------------------------------------------------------|-----------------|
| \xtoksapp | 676, 679, 1055, |
| 1419, 1779, 1823, 1864, 1870, 1988, 1993, 2802, 3106 | |