

# The documented source of Memoize, Advice and CollArgs

Memoize v1.1.2, Advice v1.1.0, CollArgs v1.1.0  
January 21, 2024

Sašo Živanović

✉ [saso.zivanovic@guest.arnes.si](mailto:saso.zivanovic@guest.arnes.si)  
📧 [spj.ff.uni-lj.si/zivanovic](mailto:spj.ff.uni-lj.si/zivanovic)  
🌐 [github.com/sasozivanovic](https://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  $\ConTeXt$  (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  $\ConTeXt$ . Even worse,  $\ConTeXt$  code is often just the same as plain  $\TeX$  code, even in cases where I’m sure  $\ConTeXt$  offers the relevant tools. This nicely proves that I have no clue about  $\ConTeXt$ . If you are willing to  $\ConTeXt$ -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}{nommz,latex}}%
  \file{nomemoize.tex}{\from{memoize.dtx}{nommz,plain}}%
  \file{t-nomemoize.tex}{\from{memoize.dtx}{nommz,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}}%
}
```

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

<b>1</b>	<b>First things first</b>	<b>3</b>
<b>2</b>	<b>The basic configuration</b>	<b>7</b>
<b>3</b>	<b>Memoization</b>	<b>11</b>
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
<b>4</b>	<b>Extraction</b>	<b>38</b>
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 <sub>E</sub> X-based extraction . . . . .	44
4.3.1	memoize-extract-one.tex . . . . .	46
<b>5</b>	<b>Automemoization</b>	<b>48</b>
5.1	L <sup>A</sup> T <sub>E</sub> X-specific handlers . . . . .	52
<b>6</b>	<b>Support for various classes and packages</b>	<b>54</b>
6.1	TikZ . . . . .	54
6.2	Forest . . . . .	54
6.3	Beamer . . . . .	55
6.4	Morewrites . . . . .	55
6.5	Biblatex . . . . .	56
<b>7</b>	<b>Initialization</b>	<b>57</b>
<b>8</b>	<b>Auxiliary packages</b>	<b>59</b>
8.1	Extending commands and environments with Advice . . . . .	59
8.1.1	Installation into a keypath . . . . .	59
8.1.2	Submitting a command or environment . . . . .	62
8.1.3	Executing a handled command . . . . .	68
8.1.4	Environments . . . . .	71
8.1.5	Error messages . . . . .	73
8.1.6	Tracing . . . . .	73
8.1.7	The TikZ collector . . . . .	75
8.2	Argument collection with CollArgs . . . . .	76
8.2.1	The keys . . . . .	78
8.2.2	The central loop . . . . .	83
8.2.3	Auxiliary macros . . . . .	86
8.2.4	The handlers . . . . .	90
8.2.5	The verbatim modes . . . . .	108
8.2.6	Transition between the verbatim and the non-verbatim mode . . . . .	113
<b>9</b>	<b>The scripts</b>	<b>123</b>
9.1	The Perl extraction script memoize-extract.pl . . . . .	123
9.2	The Python extraction script memoize-extract.py . . . . .	135
9.3	The Perl clean-up script memoize-clean.pl . . . . .	143
9.4	The Python clean-up script memoize-clean.py . . . . .	146

# 1 First things first

Identification of memoize, memoizable and nomemoize.

```
1 (*mmz)
2 <latex>\ProvidesPackage{memoize}[2024/01/21 v1.1.2 Fast and flexible externalization]
3 <context>%D \module[
4 <context>%D     file=t-memoize.tex,
5 <context>%D     version=1.1.2,
6 <context>%D     title=Memoize,
7 <context>%D     subtitle=Fast and flexible externalization,
8 <context>%D     author=Saso Zivanovic,
9 <context>%D     date=2024-01-21,
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 2024/01/21 v1.1.2
17 </mmz>
18 (*mmzable)
19 <latex>\ProvidesPackage{memoizable}[2024/01/21 v1.1.2 A programmer's stub for Memoize]
20 <context>%D \module[
21 <context>%D     file=t-memoizable.tex,
22 <context>%D     version=1.1.2,
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=2024-01-21,
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 2024/01/21 v1.1.2
34 </mmzable>
35 (*nommz)
36 <latex>\ProvidesPackage{nomemoize}[2024/01/21 v1.1.2 A no-op stub for Memoize]
37 <context>%D \module[
38 <context>%D     file=t-nomemoize.tex,
39 <context>%D     version=1.1.2,
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=2024-01-21,
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 2024/01/21 v1.1.2
51 </nommz>
```

Required packages and L<sup>A</sup>T<sub>E</sub>Xization of plain T<sub>E</sub>X and ConT<sub>E</sub>Xt.

```
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

```

`pdftexcmds.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 `pdftexcmds.lua` and copy-paste what we need from `pdftexcmds.sty`.

```

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

```

Lua function `oberdiek.pdftexcmds.filesize` requires the `kpse` library, which is not loaded in `ConTeXt`, see [github.com/latex3/lua-uni-algos/issues/3](https://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<sub>E</sub>Xt, `\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<sub>E</sub>Xt 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<sub>E</sub>X. 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 (<sup>M</sup>§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 TeX file #2 into token register #1 (under the current category code regime); `\toksapp` is defined in CollArgs.

```

214 \def\filetotoks#1#2{%
215   \immediate\mmz@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 }

```

`\mmz@openin` A workaround for morewrites.

```

\mmz@openout
224 \let\mmz@openin\openin
225 \let\mmz@openout\openout

```

Other little things.

```

226 \newif\ifmmz@temp
227 \newtoks\mmz@temptoks
228 \newbox\mmz@box
229 \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.

```
230 \def\mmzset#1{\pgfkeys{/mmz}{#1}\ignorespaces}
```

```

231 </mmz>
232 < *nommz, mmzable & generic>
233 \def\mmzset#1{\ignorespaces}
234 </nommz, mmzable & generic>

```

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

```

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

```

**enable** These keys set TeX-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 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<sup>A</sup>T<sub>E</sub>X; 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.

```

240 < *mmz, nommz, mmzable & generic>
241 \newif\ifmemoize
242 </mmz, nommz, mmzable & generic>
243 < *mmz>
244 \mmzset{%
245   enable/.style={begindocument/.append code=\memoizetrue},
246   disable/.style={begindocument/.append code=\memoizefalse},
247   begindocument/.append style={
248     enable/.code=\memoizetrue,
249     disable/.code=\memoizefalse,
250   },

```

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

```

251   enable,

```

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

```

252   options/.style={#1},
253 }

```

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

```

254 \def\mmz@mode@normal{0}
255 \def\mmz@mode@readonly{1}
256 \def\mmz@mode@recompile{2}
257 \let\mmz@mode\mmz@mode@normal
258 \mmzset{%
259   normal/.code={\let\mmz@mode\mmz@mode@normal},
260   readonly/.code={\let\mmz@mode\mmz@mode@readonly},
261   recompile/.code={\let\mmz@mode\mmz@mode@recompile},
262 }

```

---

<sup>1</sup>In fact, this code treats anything but 1 and 2 as normal.



**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`.

```
263 \mmzset{%
264   prefix/.code={\mmz@parse@prefix{#1}},
265 }
```

**\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.

```
266 \begingroup
267 \catcode`\/=12
268 \gdef\mmz@parse@prefix#1{%
269   \edef\mmz@prefix{\detokenize\expandafter{\expanded{#1}}}%
270   \def\mmz@prefix@dir{%
271     \def\mmz@prefix@name{%
272       \expandafter\mmz@parse@prefix@i\mmz@prefix/\mmz@eov
273     }
274   \gdef\mmz@parse@prefix@i#1/#2{%
275     \ifx\mmzeov#2%
276       \def\mmz@prefix@name{#1}%
277     \else
278       \appto\mmz@prefix@dir{#1/}%
279       \expandafter\mmz@parse@prefix@i\expandafter#2%
280     \fi
281   }
282 \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.

```
283 \mmzset{%
284   begindocument/.append style={
285     prefix/.append code=\mmz@maybe@mkmemodir\mmz@record@prefix,
286   },
```

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.

```
287   begindocument/.append code=\mmz@maybe@mkmemodir,
288 }
```

**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`.

```
289 \mmzset{
290   mkdir/.is if=mmz@mkdir,
291   mkdir command/.store in=\mmz@mkdir@command,
292   mkdir command={},
293 }
```

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`.

```
294 \newif\ifmmz@mkdir
295 \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).

```
296 \def\mmz@maybe@mkmemodir{%
297   \ifmmz@mkdir
298     \ifdefempty\mmz@mkdir@command{}{%
299       \ifdefempty\mmz@prefix@dir{}{%
300         \mmz@remove@quotes{\mmz@prefix@dir}\mmz@temp
301         \pdf@system{\mmz@mkdir@command\space"\mmz@temp"}%
302       }%
303     }%
304   \fi
305 }
```

`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.

```
306 \mmzset{%
307   memo dir/.style={prefix={#1.memo.dir/}},
308   memo dir/.default=\jobname,
309   no memo dir/.style={prefix={#1.}},
310   no memo dir/.default=\jobname,
311   no memo dir,
312 }
```

`\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.

```
313 \def\mmz@remove@quotes#1#2{%
314   \def\mmz@remove@quotes@end{\let#2\mmz@temp}%
315   \def\mmz@temp{}%
316   \expanded{%
317     \noexpand\mmz@remove@quotes@i
318     \detokenize\expandafter{\expanded{#1}}%
319     "\noexpand\mmz@eov
320   }%
321 }
322 \def\mmz@remove@quotes@i{%
323   \CollectArgumentsRaw
324   {\collargsBraceCollectedfalse
325     \collargsNoDelimiterstrue
326     \collargsAppendPostwrap{##1}}%
327   }%
328   {u"u\mmz@eov}%
329   \mmz@remove@quotes@ii
330 }
331 \def\mmz@remove@quotes@ii#1#2{%
332   \appto\mmz@temp{#1}%
333   \ifx&#2&%
334     \mmz@remove@quotes@end
335     \expandafter\@gobble
```

```

336 \else
337   \expandafter\@firstofone
338 \fi
339 {\mmz@remove@quotes@i#2\mmz@eov}%
340 }

```

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

```

341 \newif\ifmmz@ignorespaces
342 \mmzset{
343 ignore spaces/.is if=mmz@ignorespaces,
344 }

```

**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.

```

345 \newif\ifmmz@verbatim
346 \def\mmzRawCollectorOptions{}
347 \mmzset{
348 verbatim/.code={%
349   \def\mmzRawCollectorOptions{\collargsVerbatim}%
350   \mmz@verbatimtrue
351 },
352 verb/.code={%
353   \def\mmzRawCollectorOptions{\collargsVerb}%
354   \mmz@verbatimtrue
355 },
356 no verbatim/.code={%
357   \def\mmzRawCollectorOptions{\collargsNoVerbatim}%
358   \mmz@verbatimfalse
359 },
360 }

```

## 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`.

```

361 \protected\def\mmz{\futurelet\mmz@temp\mmz@i}
362 \def\mmz@i{%

```

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

```

363 \begingroup

```

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

```

364 \ifx\mmz@temp[%
365   \def\mmz@verbatim@fix{}%
366   \expandafter\mmz@ii
367 \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.)

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

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

```
369 \expandafter\mmz@iii
370 \fi
371 }
372 \def\mmz@ii[#1]{%
```

Apply the options given in the optional argument.

```
373 \mmzset{#1}%
374 \mmz@iii
375 }
376 \def\mmz@iii{%
```

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

```
377 \ifmmz@verbatim
378 \expandafter\mmz@do@verbatim
379 \else
380 \expandafter\mmz@do
381 \fi
382 }
```

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

```
383 \long\def\mmz@do#1{%
384 \Memoize{#1}{#1}%
385 }%
```

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).

```
386 \def\mmz@do@verbatim#1{%
387 \expanded{%
388 \noexpand\CollectArgumentsRaw{%
389 \noexpand\collargsCaller{\noexpand\mmz}%
390 \expandonce\mmzRawCollectorOptions
391 \mmz@verbatim@fix
392 }%
393 }{+m}\mmz@do
394 }
```

**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 Con $\TeX$ t environments in parallel. The definition of the plain  $\TeX$  and Con $\TeX$ t 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).

```
395 <*\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.

```
396 \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.

```
397 \end{memoize}%
```

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

```
398 \begingroup
```

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.

```
399 \def\mmz@temp{#1}%
400 \ifx\mmz@temp\mmz@noarg
401   \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
402 \else
403   \def\mmz@verbatim@fix{}%
404   \mmzset{#1}%
405 \fi
406 \mmz@env@iii
407 }{}
408 \def\mmz@noarg{\mmz@noarg}
409 </latex>
410 <plain>\def\memoize{%
411 <context>\def\startmemoize{%
412   <*plain, context>
413   \begingroup
```

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

```
414 \futurelet\mmz@temp\mmz@env@i
415 }
416 \def\mmz@env@i{%
417 \ifx\mmz@temp[%]
418   \def\mmz@verbatim@fix{}%
419   \expandafter\mmz@env@ii
420 \else
421   \def\mmz@verbatim@fix{\noexpand\collargsFixFromNoVerbatim}%
422   \expandafter\mmz@env@iii
423 \fi
424 }
425 \def\mmz@env@ii[#1]{%
426 \mmzset{#1}%
427 \mmz@env@iii
428 }
429 </plain, context>
430 \def\mmz@env@iii{%
431 \long\edef\mmz@do##1{%
```

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

```
432 \noexpand\Memoize{##1}{##1\unskip}%
433 }%
434 \expanded{%
435 \noexpand\CollectArgumentsRaw%
```

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

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

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

```
437 \expandonce\mmzRawCollectorOptions
```

The category code fix, if needed.

```
438 \ifmmz@verbatim\mmz@verbatim@fix\fi
439 }%
```

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

```
440 }{&&{\collargsAppendPostwrap{}}!t{ }+b{memoize}}{\mmz@do}%
441 }%
442 </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`.

```
443 <*mmz, nommz>
444 \protected\def\nommz#1#{%
445 \afterassignment\nommz@i
446 \let\mmz@temp
447 }
448 \def\nommz@i{%
449 \bgroup
450 \memoizefalse
451 }
452 <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.

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

Start a group to delimit `\memoizefalse`.

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

No optional argument, no problems with spaces.

```

471 \fi
472 }
473 \def\nommz@env@ii[#1]{%
474 \ignorespaces
475 }
476 <plain>\def\endnomemoize{%
477 <context>\def\stopnomemoize{%
478 \endgroup
479 \unskip
480 }
481 </plain,context>
482 <*nommz>
483 <plain,latex>\let\memoize\nomemoize
484 <plain,latex>\let\endmemoize\endnomemoize
485 <context>\let\startmemoize\startnomemoize
486 <context>\let\stopmemoize\stopnomemoize
487 </nommz>
488 </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.

```

489 <*mmz,nommz,mmzable & generic>
490 \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.

```

491 \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.

```

492 </mmz,nommz,mmzable & generic>
493 <*mmz>
494 \def\mmz@maybe@scantokens{%
495 \ifmmz@verbatim
496 \expandafter\mmz@scantokens
497 \else
498 \expandafter\@firstofone
499 \fi
500 }

```

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.)

```

501 \long\def\mmz@scantokens#1{%
502 \expanded{%
503 \newlinechar=13
504 \unexpanded{\scantokens{#1\endinput}}%
505 \newlinechar=\the\newlinechar
506 }%
507 }

```

`\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 (<sup>M</sup>§4.1).

`\Memoize` takes two arguments, which contain two potentially different versions of the code submitted to memoization: `#1` contains the code which *(code MD5 sum)* 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.

```
508 \newtoks\mmz@mdfive@source
509 \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.

```
510 \let\Memoize\@secondoftwo
511 \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`).

```
512 \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{...}`.

```
513 \expandafter\expandafter\expandafter\expandafter
514 \expandafter\expandafter\expandafter
515 \mmz@exec@source
516 \expandafter\expandafter\expandafter\expandafter
517 \expandafter\expandafter\expandafter
518 {%
519 \mmz@maybe@scantokens{#2}%
520 }%
521 \mmz@trace@Memoize
```

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

```
522 \let\mmz@action\mmz@compile
```

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

```
523 \ifmemoizing
524 \else
525 \ifmemoize
```

Compute *(code md5sum)* 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.

```
526 \xdef\mmz@code@mdfivesum{\pdf@mdfivesum{\the\mmz@mdfive@source}}%
527 \mmz@trace@code@mdfive
```



Recompile mode forces memoization.

```

528     \ifnum\mmz@mode=\mmz@mode@recompile\relax
529         \ifnum\pdf@draftmode=0
530             \let\mmz@action\mmz@memoize
531         \fi
532     \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.

```

533         \mmz@process@cmemo
534         \ifmmzUnmemoizable
535             \mmz@trace@cmemo@unmemoizable
536         \else
537             \ifmmz@abort

```

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

```

538             \mmz@trace@process@cmemo@fail
539             \ifnum\mmz@mode=\mmz@mode@readonly\relax
540         \else
541             \ifnum\pdf@draftmode=0
542                 \let\mmz@action\mmz@memoize
543             \fi
544         \fi
545     \else
546         \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.

```

547         \let\mmz@action\mmz@process@ccmemo
548     \fi
549 \fi
550 \fi
551 \fi
552 \fi
553 \mmz@action
554 }

```

`\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`.

```

555 \def\mmz@compile{%
556     \mmz@trace@compile
557     \expanded{%
558         \endgroup

```

```

559 \noexpand\inmemoizetrue
560 \the\mmz@exec@source
561 \ifinmemoize\noexpand\inmemoizetrue\else\noexpand\inmemoizefalse\fi
562 }%
563 }

```

`abortOnError` In Lua<sub>TEX</sub>, we can check 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 Con<sub>TEX</sub>t, as it appears to stop on any error?

```

564 <!*context>
565 \ifdefined\luatexversion
566 \directlua{%
567   luatexbase.add_to_callback(
568     "show_error_message",
569     function()
570       memoize.abort = true
571       texio.write_nl(status.lasterrorstring)
572     end,
573     "Abort memoization on error"
574   )
575 }%
576 \def\mmz@lua@atbeginmemoization{%
577   \directlua{memoize.abort = false}%
578 }%
579 \def\mmz@lua@atendmemoization{%
580   \directlua{%
581     if memoize.abort then
582       tex.print("\noexpand\mmzAbort")
583     end
584   }%
585 }%
586 \else
587 </!*context>
588 \let\mmz@lua@atbeginmemoization\relax
589 \let\mmz@lua@atendmemoization\relax
590 <!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.

```

591 \def\mmz@memoize{%
592   \mmz@trace@memoize
593   \memoizingtrue
594   \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.

```

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

```

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

```
607 \mmzAtBeginMemoization
608 \mmzDriver{\the\mmz@exec@source}%
609 \mmzAtEndMemoization
610 \mmzAtEndMemoizationExtra
611 \mmz@lua@atendmemoization
612 \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.

```
613 \global\mmzContextExtra{}%
614 \gtoksapp\mmzCMemo{\global\mmzUnmemoizabletrue}%
615 \mmz@write@cmemo
616 \mmz@trace@endmemoize@unmemoizable
617 \PackageInfo{memoize}{Marking this code as unmemoizable}%
618 \else
619 \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.

```
620 \mmz@trace@endmemoize@aborted
621 \PackageInfo{memoize}{Memoization was aborted}%
622 \mmz@compute@context@mdfivesum
623 \mmz@write@cmemo
624 \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).

```
625 \mmz@compute@context@mdfivesum
626 \mmz@write@cmemo
627 \mmz@write@ccmemo
628 \mmz@shipout@externs
629 \mmz@trace@endmemoize@ok
630 \fi
631 \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.

```
632 \expandafter\endgroup
633 \expandafter\let
634 \expandafter\mmzIncludeExtern\expandafter\mmz@include@extern@from@tbe@box
635 \mmzAfterMemoization
636 \mmzAfterMemoizationExtra
637 }
```

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

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

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

```
639 \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).

```
640 \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`.

```
641 \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`.

```
642 \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 `at end memoization` tra macro (`\mmzAtEndMemoizationExtra`, `\mmzAfterMemoizationExtra`; there is no `after memoization` `\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.

```
643 \def\mmzAtBeginMemoization{}
644 \def\mmzAtEndMemoization{}
645 \def\mmzAfterMemoization{}
646 \mmzset{
647   at begin memoization/.code={%
648     \ifmemoizing
649       \expandafter\@firstofone
650     \else
651       \expandafter\appto\expandafter\mmzAtBeginMemoization
652     \fi
653     {#1}%
654   },
655   at end memoization/.code={%
656     \ifmemoizing
657       \expandafter\gappto\expandafter\mmzAtEndMemoizationExtra
658     \else
659       \expandafter\appto\expandafter\mmzAtEndMemoization
660     \fi
661     {#1}%
662   },
663   after memoization/.code={%
664     \ifmemoizing
665       \expandafter\gappto\expandafter\mmzAfterMemoizationExtra
666     \else
667       \expandafter\appto\expandafter\mmzAfterMemoization
668     \fi
669     {#1}%
670   },
671 }
```

`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.

```

672 \mmzset{
673   driver/.store in=\mmzDriver,
674   driver=\mmzSingleExternDriver,
675 }

```

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

```

676 \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.)

```

677 \long\def\mmzSingleExternDriver#1{%
678   \xtoksapp\mmzCCMemo{\mmz@maybe@quitvmode}%
679   \setbox\mmz@box\mmz@capture{#1}%
680   \mmzExternalizeBox\mmz@box\mmz@temptoks
681   \xtoksapp\mmzCCMemo{\the\mmz@temptoks}%
682   \mmz@maybe@quitvmode\box\mmz@box
683 }

```

`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.

```

684 \mmzset{
685   capture/.is choice,
686   capture/hbox/.code={%
687     \let\mmz@capture\hbox
688     \def\mmz@maybe@quitvmode{\noexpand\quitvmode}%
689   },
690   capture/vbox/.code={%
691     \let\mmz@capture\vbox
692     \def\mmz@maybe@quitvmode{}%
693   },
694   capture=hbox,
695 }

```

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/aut`. Given `\langle suffix \rangle` as the only argument, it declares conditional `\ifmemoizing\langle suffix \rangle`, 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.

```

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

```

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

```

701     \@firstofone
702   }%
703 },
704 }

```

**\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.

```

705 \newcommand\IfMemoizing[2][\mmz@Ifmemoizing@nogrouplevel]{%>\fi
706 \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.

```

707 \ifnum\currentgrouplevel=\the\numexpr\memoizinggrouplevel+#1\relax\relax
708   \expandafter\expandafter\expandafter\@firstoftwo
709 \else
710   \expandafter\expandafter\expandafter\@secondoftwo
711 \fi
712 \else
713   \expandafter\@secondoftwo
714 \fi
715 }
716 \def\mmz@Ifmemoizing@nogrouplevel{0\relax\relax\fi\iftrue}

```

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

```

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

```

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

```

741 \def\mmz@trace@process@cmemo@on{\mmz@trace{\space\space
742   Attempting to utilize c-memo \mmz@cmemo@path}}

```

```

743 \def\mmz@trace@process@no@cmemo@on{\mmz@trace{\space\space
744   C-memo does not exist}}
745 \def\mmz@trace@process@cmemo@ok@on{\mmz@trace{\space\space
746   C-memo was processed successfully}\mmz@trace@context}
747 \def\mmz@trace@process@cmemo@fail@on{\mmz@trace{\space\space
748   C-memo input failed}}
749 \def\mmz@trace@cmemo@unmemoizable@on{\mmz@trace{\space\space
750   This code was marked as unmemoizable}}
751 \def\mmz@trace@process@ccmemo@on{\mmz@trace{\space\space
752   Attempting to utilize cc-memo \mmz@ccmemo@path\space
753   (\ifmmz@direct@ccmemo@input\else in\fi direct input)}}
754 \def\mmz@trace@resource@on#1{\mmz@trace{\space\space
755   Extern file does not exist: #1}}
756 \def\mmz@trace@process@ccmemo@ok@on{%
757   \mmz@trace{\space\space Utilization successful}}
758 \def\mmz@trace@process@no@ccmemo@on{%
759   \mmz@trace{\space\space CC-memo does not exist}}
760 \def\mmz@trace@process@ccmemo@fail@on{%
761   \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.

```

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

```

```

801 \let\mmz@trace@cmemo@unmemoizable\relax
802 \let\mmz@trace@process@ccmemo\relax
803 \let\mmz@trace@resource\@gobble
804 \let\mmz@trace@process@ccmemo@ok\relax
805 \let\mmz@trace@process@no@ccmemo\relax
806 \let\mmz@trace@process@ccmemo@fail\relax
807 }
808 \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`.

```

809 \newtoks\mmzContext
810 \newtoks\mmzContextExtra

```

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

```

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

```

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

```

818   {#1,}%
819 },
820 clear context/.code={%
821   \ifmemoizing
822     \expandafter\global\expandafter\mmzContextExtra
823   \else
824     \expandafter\mmzContext
825   \fi
826   {}}%
827 },
828 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 829 meaning to context/.code={\forcsvlist\mmz@mtoc{#1}},
key value to context 830 csname meaning to context/.code={\mmz@mtoc@csname{#1}},
/handlers/.meaning to context 831 key meaning to context/.code={%
/handlers/.value to context 832   \forcsvlist\mmz@mtoc\mmz@mtoc@keycmd{#1}},
833 key value to context/.code={\forcsvlist\mmz@mtoc@key{#1}},
834 /handlers/.meaning to context/.code={\expanded{%
835   \noexpand\mmz@mtoc@csname{pgfk@\pgfkeyscurrentpath/.@cmd}}},
836 /handlers/.value to context/.code={%
837   \expanded{\noexpand\mmz@mtoc@csname{pgfk@\pgfkeyscurrentpath}}},
838 }

```

```

839 \def\mmz@mtoc#1{%
840   \collargs@cs@cases{#1}%
841   {\mmz@mtoc@cmd{#1}}%
842   {\mmz@mtoc@error@notcsorenv{#1}}%
843   {%
844     \mmz@mtoc@csname{%
845 <context>       start%
846                 #1}%

```



```

847         \mmz@mtoc@csname{%
848 <latex, plain>         end%
849 <context>             stop%
850                       #1}%
851     }%
852 }
853 \def\mmz@mtoc@cmd#1{%
854   \begingroup
855   \escapechar=-1
856   \expandafter\endgroup
857   \expandafter\mmz@mtoc@csname\expandafter{\string#1}%
858 }
859 \def\mmz@mtoc@csname#1{%
860   \pgfkeysvalueof{/mmz/context/.@cmd}%
861   \detokenize{#1}={\expandafter\meaning\csname#1\endcsname}%
862   \pgfeov
863 }
864 \def\mmz@mtoc@key#1{\mmz@mtoc@csname{pgfk@#1}}
865 \def\mmz@mtoc@keycmd#1{\mmz@mtoc@csname{pgfk@#1/.@cmd}}
866 \def\mmz@mtoc@error@notcsorenv#1{%
867   \PackageError{memoize}{'\detokenize{#1}' passed to key 'meaning to context'
868     is neither a command nor an environment}{}%
869 }

```

### 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`.

```
870 \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.

```
871 \newtoks\mmzCMemo
```

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

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

```

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

```

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

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

Open the file for writing.

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

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

```
879   \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.

```

880   \immediate\write\mmz@out{%
881     \global\mmzContextExtra{\the\mmzContextExtra}\collargs@percentchar
882   }%

```

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

```
883   \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.

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

Close the file.

```
888 \immediate\closeout\mmz@out
```

Record that we wrote a new c-memo.

```
889 \pgfkeysalso{/mmz/record/new cmemo={\mmz@cmemo@path}}%
890 }
```

`\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.

```
891 \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.

```
892 \def\mmz@process@cmemo{%
893   \mmz@trace@process@cmemo
```

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

```
894   \global\mmz@aborttrue
```

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

```
895   \global\mmzUnmemoizablefalse
896   \def\mmzMemo{\global\mmz@abortfalse}%
```

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

```
897   \global\mmzContextExtra{}
```

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

```
898   \IfFileExists{\mmz@cmemo@path}{%
899     \input{\mmz@cmemo@path}%
900     \pgfkeysalso{/mmz/record/used cmemo={\mmz@cmemo@path}}%
901   }-%
902   \mmz@trace@process@no@cmemo
903 }
```

Compute the context MD5 sum.

```
904   \mmz@compute@context@mdfivesum
905 }
```

`\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.

```
906 \def\mmz@compute@context@mdfivesum{%
907   \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.

```

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

```

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

```

916 <latex> \protected@xdef
917 <!latex> \xdef
918 \mmz@context@key{\mmz@context@key}%
919 \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`.

```

920 \xdef\mmz@context@mdfivesum{\pdf@mdfivesum{\expandonce\mmz@context@key}}%
921 }

```

### 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`.

```

922 \def\mmz@ccmemo@path{%
923 \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.

```

924 \newtoks\mmzCCMemo

```

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

```

\ifmmz@include@context

```

```

925 \newif\ifmmz@include@context
926 \mmzset{%
927 include context in ccmemo/.is if=mmz@include@context,
928 }

```

`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.

```
929 \newif\ifmmz@direct@ccmemo@input
930 \mmzset{%
931   direct ccmemo input/.is if=mmz@direct@ccmemo@input,
932 }
```

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

```
933 \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`.

```
934   \immediate\mmz@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.

```
935   \begingroup
936   \the\mmz@ccmemo@resources
937   \endgroup
```

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

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

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

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

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

```
944   \immediate\write\mmz@out{\noexpand\mmzEndMemo}%
945   \immediate\closeout\mmz@out
```

Record that we wrote a new cc-memo.

```
946   \pgfkeysalso{/mmz/record/new ccmemo={\mmz@ccmemo@path}}%
947 }
```

`\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.

```
948 \def\mmz@ccmemo@append@resource#1{%
949   \mmz@seq=#1\relax
950   \immediate\write\mmz@out{%
951     \string\mmzResource{\mmz@extern@name}\collargs@percentchar}%
952 }
```

`\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.

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

We check for existence using `\pdffilesize`, because an empty PDF, which might be produced by a failed T<sub>E</sub>X-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.

```
954 \ifnum0\pdf@filesize{\mmz@prefix@dir#1}=0
955   \ifmmz@direct@ccmemo@input
956     \let\mmzMemo\endinput
957   \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).

```
958     \long\def\mmzMemo###1\mmzEndMemo\par{%
959     \fi
960     \mmz@trace@resource{#1}%
961     \fi
962 }
```

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

`\mmzThisContext`

`\mmzEndMemo`

```
963 \def\mmz@process@ccmemo{%
964   \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!

```
965 \global\mmz@aborttrue
```

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

```
966 \def\mmzMemo{%
967   \endgroup
968   \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.

```
969 \let\mmzIncludeExtern\mmz@include@extern
970 }%
```

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.

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

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

```
975   \unexpanded{%
```

```

976     \def\mmz@temp\par{}%
977     \mmz@temp
978   }%
979   \fi
980 }%

```

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).

```

981 \xdef\mmzThisContext{%
982   \ifmmz@direct@ccmemo@input
983     \noexpand\endinput
984   \else
985     \unexpanded{%
986       \long\def\mmz@temp##1\mmzEndMemo\par{}%
987       \mmz@temp
988     }%
989   \fi
990 }%

```

Input the cc-memo if it exists.

```

991 \IfFileExists{\mmz@ccmemo@path}{%
992   \ifmmz@direct@ccmemo@input
993     \input{\mmz@ccmemo@path}%
994   \else

```

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

```

995   \filetotoks\toks@{\mmz@ccmemo@path}%
996   \the\toks@
997   \fi

```

Record that we have used the cc-memo.

```

998   \pgfkeysalso{/mmz/record/used ccmemo={\mmz@ccmemo@path}}%
999 }{%
1000   \mmz@trace@process@no@ccmemo
1001 }%
1002 \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`.)

```

1003   \mmz@trace@process@ccmemo@fail
1004   \ifnum\mmz@mode=\mmz@mode@readonly\relax
1005     \expandafter\expandafter\expandafter\mmz@compile
1006   \else
1007     \expandafter\expandafter\expandafter\mmz@memoize
1008   \fi
1009 \else
1010   \mmz@trace@process@ccmemo@ok
1011 \fi
1012 }

```

### 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.

```

1013 \newcount\mmz@seq
1014 \def\mmz@extern@basename{%
1015   \mmz@prefix@name\mmz@code@mdfivesum-\mmz@context@mdfivesum
1016   \ifnum\mmz@seq>0 -\the\mmz@seq\fi
1017 }
1018 \def\mmz@extern@name{\mmz@extern@basename.pdf}
1019 \def\mmz@extern@basepath{\mmz@prefix@dir\mmz@extern@basename}
1020 \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`

```

1021 \mmzset{
1022   padding left/.store in=\mmz@padding@left,
1023   padding right/.store in=\mmz@padding@right,
1024   padding top/.store in=\mmz@padding@top,
1025   padding bottom/.store in=\mmz@padding@bottom,

```

`padding` A shortcut for setting all four paddings at once.

```

1026   padding/.style={
1027     padding left=#1, padding right=#1,
1028     padding top=#1, padding bottom=#1
1029   },

```

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.

```

1030   padding=1in,

```

`padding to context` This key adds padding to the context. Note that we add the padding expression (`\mmz@padding`s, 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.

```

1031   padding to context/.style={
1032     context={padding=(\mmz@padding)},
1033   },

```

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.

```

1034   padding to context,
1035 }
1036 \def\mmz@padding{%
1037   \mmz@padding@left,\mmz@padding@bottom,\mmz@padding@right,\mmz@padding@top
1038 }

```

`\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*).

```
1039 \def\mmzExternalizeBox#1#2{%
1040   \begingroup
```

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

```
1041   \def\width{\wd#1 }%
1042   \def\height{\ht#1 }%
1043   \def\depth{\dp#1 }%
```

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

```
1044   \xdef\mmz@global@temp{%
```

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

```
1045     \noexpand\mmzIncludeExtern
```

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

```
1046     {\the\mmz@seq}%
```

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

```
1047     \ifhbox#1\noexpand\hbox\else\noexpand\vbox\fi
```

The dimensions of the extern.

```
1048     {\the\wd#1}%
1049     {\the\ht#1}%
1050     {\the\dp#1}%
```

The padding values.

```
1051     {\the\dimexpr\mmz@padding@left}%
1052     {\the\dimexpr\mmz@padding@bottom}%
1053     {\the\dimexpr\mmz@padding@right}%
1054     {\the\dimexpr\mmz@padding@top}%
1055   }%
```

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.

```
1056   \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.

```
1057   \xtoksapp\mmz@ccmemo@resources{%
1058     \noexpand\mmz@ccmemo@append@resource{\the\mmz@seq}%
1059   }%
```

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

```
1060   \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>`.

```
1061 \endgroup
1062 #2\expandafter{\mmz@global@temp}%
1063 }
```

`\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.

```
1064 \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.

```
1065 \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`,<sup>2</sup> 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.

```
1066 \def\mmz@shipout@externs{%
1067 \global\mmz@seq 0
1068 \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`).

```
1069 \def\width{\wd\mmz@box}%
1070 \def\height{\ht\mmz@box}%
1071 \def\depth{\dp\mmz@box}%
1072 \vskip1pt
1073 \ifmmzkeepexterns\expandafter\unvcopy\else\expandafter\unvbox\fi\mmz@tbe@box
1074 \@whilesw\ifdimOpt=\lastskip\fi{%
1075 \setbox\mmz@box\lastbox
1076 \mmz@shipout@extern
1077 }%
1078 }%
1079 }
```

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

```
1080 \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.

```
1081 \edef\expectedwidth{\the\dimexpr
1082 (\mmz@padding@left) + \wd\mmz@box + (\mmz@padding@right)}%
1083 \edef\expectedheight{\the\dimexpr
1084 (\mmz@padding@top) + \ht\mmz@box + \dp\mmz@box + (\mmz@padding@bottom)}%
```

<sup>2</sup>The looping code is based on TeX.SE answer [tex.stackexchange.com/a/25142/16819](https://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`.

```

1085 \begingroup
1086 \ifnum\mag=1000
1087 \else
1088   \mmz@shipout@mag
1089 \fi

```

Setup the geometry of the extern page. In plain  $\TeX$  and  $\LaTeX$ , setting `\pdfpagewidth` and `\pdfpageheight` seems to do the trick of setting the extern page dimensions. In  $\ConTeXt$ , 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.

```

1090 <*plain, latex>
1091 \pdfpagewidth\dimexpr
1092   (\mmz@padding@left) + \wd\mmz@box + (\mmz@padding@right)\relax
1093 \pdfpageheight\dimexpr
1094   (\mmz@padding@top) + \ht\mmz@box + \dp\mmz@box+ (\mmz@padding@bottom)\relax
1095 </plain, latex>
1096 <*context>
1097 \edef\mmz@temp{%
1098   \noexpand\directlua{
1099     backends.codeinjections.setupcanvas({
1100       paperwidth=\the\numexpr\pagewidth,
1101       paperheight=\the\numexpr\pageheight
1102     })
1103   }%
1104 }%
1105 \expanded{%
1106   \noexpand\directlua{
1107     backends.codeinjections.setupcanvas({
1108       paperwidth=\the\numexpr\dimexpr
1109         \mmz@padding@left + \wd\mmz@box + \mmz@padding@right\relax,
1110       paperheight=\the\numexpr\dimexpr
1111         \mmz@padding@top + \ht\mmz@box + \dp\mmz@box+ \mmz@padding@bottom\relax
1112     })
1113   }%
1114 }%
1115 </context>

```

We complete the page setup by setting the content offset.

```

1116 \hoffset\dimexpr\mmz@padding@left - \pdfhorigin\relax
1117 \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,  $\LaTeX$  and  $\ConTeXt$  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  $\LaTeX$ -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.

```

1118 <latex> \let\protect\noexpand
1119 \pdf@primitive\shipout\box\mmz@box
1120 <context> \mmz@temp
1121 \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.

```
1122 \global\advance\mmzExternPages1
```

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

```
1123 \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.

```

1124 \edef\pagenumber{%
1125 \the\numexpr\mmzRegularPages

```

In L<sup>A</sup>T<sub>E</sub>X, the `\mmzRegularPages` holds to number of pages already shipped out. In ConT<sub>E</sub>Xt, the counter is already increased while processing the page, so we need to subtract 1.

```

1126 <context> -1%
1127 +\mmzExternPages+\mmzExtraPages
1128 }%

```

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`.

```
1129 \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`.

```

1130 \global\advance\mmz@seq1
1131 }

```

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

```
1132 \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  $(ax + cy + e, bx + dy + f)$ .)

```

1133 \setbox\mmz@box\hbox{%
1134 \pdfliteral{q \mmz@inverse@mag\space 0 0 \mmz@inverse@mag\space 0 0 cm}%
1135 \copy\mmz@box\relax
1136 \pdfliteral{Q}%
1137 }%

```

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

```

1138 \dimen0=\dimexpr\mmz@padding@left\relax
1139 \edef\mmz@padding@left{\the\dimexpr\mmz@inverse@mag\dimen0}%
1140 \dimen0=\dimexpr\mmz@padding@bottom\relax
1141 \edef\mmz@padding@bottom{\the\dimexpr\mmz@inverse@mag\dimen0}%
1142 \dimen0=\dimexpr\mmz@padding@right\relax
1143 \edef\mmz@padding@right{\the\dimexpr\mmz@inverse@mag\dimen0}%
1144 \dimen0=\dimexpr\mmz@padding@top\relax
1145 \edef\mmz@padding@top{\the\dimexpr\mmz@inverse@mag\dimen0}%

```

Scale the extern box.

```

1146 \wd\mmz@box=\mmz@inverse@mag\wd\mmz@box\relax
1147 \ht\mmz@box=\mmz@inverse@mag\ht\mmz@box\relax
1148 \dp\mmz@box=\mmz@inverse@mag\dp\mmz@box\relax
1149 }

```

`\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 `\Pgf@geT` from `pgfutil-common` (but rename it).

```

1150 {\catcode`\p=12\catcode`\t=12\gdef\mmz@Pgf@geT#1pt{#1}}
1151 \mmzset{begindocument/.append code={%
1152     \edef\mmz@inverse@mag{\expandafter\mmz@Pgf@geT\the\dimexpr 1000pt/\mag}%
1153 }}

```

`\mmzRegularPages` This counter holds the number of pages shipped out by the format's shipout routine.  $\text{\LaTeX}$  and  $\text{\ConTeXt}$  keep track of this in dedicated counters, so we simply use those. In plain  $\text{\TeX}$ , 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.

```

1154 <latex>\let\mmzRegularPages\ReadOnlyShipoutCounter
1155 <context>\let\mmzRegularPages\realpageno
1156 <plain>\newcount\mmzRegularPages

```

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

```
1157 \newcount\mmzExternPages
```

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

```

1158 \mmzset{
1159     enddocument/afterlastpage/.append code={%
1160         \ifnum\mmzExternPages>0
1161             \PackageWarning{memoize}{The compilation produced \the\mmzExternPages\space
1162                 new extern\ifnum\mmzExternPages>1 s\fi}%
1163         \fi
1164     },
1165 }

```

`\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.

```
1166 \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).

```
1167 \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`).

```
1168 \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.

```
1169 \setbox\mmz@box=#2{%
1170   \setbox0=\hbox{%
1171     \lower\dimexpr #5+#7\relax\hbox{%
1172       \hskip -#6\relax
1173       \setbox0=\hbox{%
1174         \mmz@insertpdfpage{\mmz@extern@path}{1}%
1175       }%
1176       \unhbox0
1177     }%
1178   }%
1179   \wd0 \dimexpr\wd0-#8\relax
1180   \ht0 \dimexpr\ht0-#9\relax
1181   \dp0 #5\relax
1182   \box0
1183 }%
```

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.)

```
1184 \mmz@tempfalse
1185 \mmz@if@roughly@equal{\mmz@tolerance}{#3}{\wd\mmz@box}{%
1186   \mmz@if@roughly@equal{\mmz@tolerance}{#4}{\ht\mmz@box}{%
1187     \mmz@temptrue
1188   }{}{}%
1189 \ifmmz@temp
1190 \else
1191   \mmz@use@memo@warning{\mmz@extern@path}{#3}{#4}{#5}%
1192 \fi
```

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

```
1193 \wd\mmz@box=#3\relax
1194 \ht\mmz@box=#4\relax
1195 \box\mmz@box
```

Record that we have used this extern.

```
1196 \pgfkeysalso{/mmz/record/used extern={\mmz@extern@path}}%
1197 }
```

```
1198 \def\mmz@use@memo@warning#1#2#3#4{%
1199   \PackageWarning{memoize}{Unexpected size of extern "#1";
1200     expected #2\space x \the\dimexpr #3+#4\relax,
1201     got \the\wd\mmz@box\space x \the\dimexpr\the\ht\mmz@box+\the\dp\mmz@box\relax}%
1202 }
```

`\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<sub>E</sub>Xt always uses LuaT<sub>E</sub>X.

```
1203 <latex, plain>\ifdef\luatexversion{%
```

```

1204 \def\mmz@insertpdfpage#1#2{% #1 = filename, #2 = page number
1205   \saveimageresource page #2 mediabox {#1}%
1206   \useimageresource\lastsavedimageresourceindex
1207 }%
1208 <*/latex,plain>
1209 }{%
1210 \ifdef\XeTeXversion{%
1211   \def\mmz@insertpdfpage#1#2{%
1212     \XeTeXpdffile #1 page #2 media
1213   }%
1214 }{% pdfLaTeX
1215   \def\mmz@insertpdfpage#1#2{%
1216     \pdfximage page #2 mediabox {#1}%
1217     \pdfrefximage\pdflastximage
1218   }%
1219 }%
1220 }
1221 </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}`.

```

1222 \def\mmz@include@extern@from@tbe@box#1#2#3#4#5#6#7#8#9{%
1223   \setbox0\vbox{%
1224     \@tempcnta#1\relax
1225     \vskip1pt
1226     \unvcopy\mmz@tbe@box
1227     \@whilenum\@tempcnta>0\do{%
1228       \setbox0\lastbox
1229       \advance\@tempcnta-1\relax
1230     }%
1231     \global\setbox1\lastbox
1232     \@whilesw\ifdim0pt=\lastskip\fi{%
1233       \setbox0\lastbox
1234     }%
1235     \box\mmz@box
1236   }%
1237   \box1
1238 }

```

## 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.

```

1239 \def\mmzvalueof#1{\pgfkeysvalueof{/mmz/#1}}
1240 \mmzset{
1241   extract/.estore in=\mmz@extraction@method,
1242   extract/.value required,
1243   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 **extract/python** 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).

```

1244 extract/perl/.code={%
1245   \mmz@clear@extraction@log
1246   \pdf@system{%
1247     \mmzvalueof{perl extraction command}\space
1248     \mmzvalueof{perl extraction options}%
1249   }%
1250   \mmz@check@extraction@log{perl}%
1251   \def\mmz@mkdir@command{\mmzvalueof{perl extraction command} --mkdir}%
1252 },
1253 perl extraction command/.initial=memoize-extract.pl,
1254 perl extraction options/.initial={\space
1255 <latex> -F latex
1256 <plain> -F plain
1257 <context> -F context
1258   \jobname\space
1259 },
1260 extract=perl,
1261 extract/python/.code={%
1262   \mmz@clear@extraction@log
1263   \pdf@system{%
1264     \mmzvalueof{python extraction command}\space
1265     \mmzvalueof{python extraction options}%
1266   }%
1267   \mmz@check@extraction@log{python}%
1268   \def\mmz@mkdir@command{\mmzvalueof{python extraction command} --mkdir}%
1269 },
1270 python extraction command/.initial=memoize-extract.py,
1271 python extraction options/.initial={\space
1272 <latex> -F latex
1273 <plain> -F plain
1274 <context> -F context
1275   \jobname\space
1276 },
1277 }
1278 \def\mmz@preamble@only@error{%
1279   \PackageError{memoize}{%
1280     Ignoring the invocation of "\pgfkeyscurrentkey".
1281     This key may only be executed in the preamble}{}%
1282 }

```

[The extraction log](#) — As we cannot access the exit status of a system command in T<sub>E</sub>X, we communicate with the system command via the “extraction log file,” produced by both T<sub>E</sub>X-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<sub>E</sub>X file, the idea is to simply input it after extracting each extern (for T<sub>E</sub>X-based extraction) or after the extraction of all externs (for the external scripts).

```

1283 \def\mmz@clear@extraction@log{%
1284   \begingroup
1285   \immediate\mmz@openout0{\jobname.mmz.log}%
1286   \immediate\closeout0
1287   \endgroup
1288 }

#1 is the extraction method.

1289 \def\mmz@check@extraction@log#1{%
1290   \begingroup \def\extractionmethod{#1}%
1291   \mmz@tempfalse \let\mmz@orig@endinput\endinput

```

```

1292 \def\endinput{\mmz@temptrue\mmz@orig@endinput}%
1293 \@input{\jobname.mmz.log}%
1294 \ifmmz@temp \else \mmz@extraction@error \fi \endgroup }
1295 \def\mmz@extraction@error{%
1296 \PackageError{memoize}{Extraction of externs from document
1297 "\jobname.pdf" using method "\extractionmethod" was
1298 unsuccessful}{The extraction script "\mmzvalueof{\extractionmethod\space
1299 extraction command}" wasn't executed or didn't finish execution
1300 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.

```

1301 \mmzset{
1302   record/.style={%
1303     record/begin/.append style={
1304       /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`.

```

1305     /mmz/record/#1/prefix/.try/.expanded=\mmz@prefix,
1306   },
1307   record/prefix/.append style={/mmz/record/#1/prefix/.try={##1}},
1308   record/new extern/.append style={/mmz/record/#1/new extern/.try={##1}},
1309   record/used extern/.append style={/mmz/record/#1/used extern/.try={##1}},
1310   record/new cmemo/.append style={/mmz/record/#1/new cmemo/.try={##1}},
1311   record/new ccmemo/.append style={/mmz/record/#1/new ccmemo/.try={##1}},
1312   record/used cmemo/.append style={/mmz/record/#1/used cmemo/.try={##1}},
1313   record/used ccmemo/.append style={/mmz/record/#1/used ccmemo/.try={##1}},
1314   record/end/.append style={/mmz/record/#1/end/.try},
1315 },
1316 }

```

**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.

```

1317 \mmzset{
1318   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.

```

1319     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.

```

1320     record/prefix/.code={\aftergroup\mmz@record@prefix},
1321     record/new extern/.code={},
1322     record/used extern/.code={},
1323     record/new cmemo/.code={},
1324     record/new ccmemo/.code={},
1325     record/used cmemo/.code={},
1326     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.

```
1327   record/end/.style={record/end/.code={}},
1328 }
1329 }
```

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

```
1330 \def\mmz@record@prefix{%
1331   \mmzset{/mmz/record/prefix/.expanded=\mmz@prefix}%
1332 }
```

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

```
1333 \mmzset{
1334   no record,
1335   record=mmz,
1336   begindocument/.append style={record/begin},
1337   enddocument/afterlastpage/.append style={record/end},
1338 }
```

#### 4.2.1 The `.mmz` file

Think of the `.mmz` record file as a  $\text{\TeX}$ -readable log file, which lets the extraction procedure know what happened in the previous compilation. The file is in  $\text{\TeX}$  format, so that we can trigger internal  $\text{\TeX}$ -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`).

```
1339 \mmzset{
1340   record/mmz/begin/.code={%
1341     \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  $\text{\TeX}$  is executed from; usually, this will be the directory containing the  $\text{\TeX}$  source).

```
1342     \immediate\mmz@openout\mmz@mmzout{\jobname.mmz}%
1343   },
```

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.

```
1344   record/mmz/prefix/.code={%
1345     \immediate\write\mmz@mmzout{\noexpand\mmzPrefix{#1}}%
1346   },
1347   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).

```

1348   \immediate\write\mmz@mmzout{%
1349     \noexpand\mmzNewExtern{#1}{\pagenumber}{\expectedwidth}{\expectedheight}%
1350   }%
1351 },
1352 record/mmz/new cmemo/.code={%
1353   \immediate\write\mmz@mmzout{\noexpand\mmzNewCMemo{#1}}%
1354 },
1355 record/mmz/new ccmemo/.code={%
1356   \immediate\write\mmz@mmzout{\noexpand\mmzNewCCMemo{#1}}%
1357 },
1358 record/mmz/used extern/.code={%
1359   \immediate\write\mmz@mmzout{\noexpand\mmzUsedExtern{#1}}%
1360 },
1361 record/mmz/used cmemo/.code={%
1362   \immediate\write\mmz@mmzout{\noexpand\mmzUsedCMemo{#1}}%
1363 },
1364 record/mmz/used ccmemo/.code={%
1365   \immediate\write\mmz@mmzout{\noexpand\mmzUsedCCMemo{#1}}%
1366 },
1367 record/mmz/end/.code={%

```

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

```

1368   \immediate\write\mmz@mmzout{\noexpand\endinput}%
1369   \immediate\closeout\mmz@mmzout
1370 },

```

#### 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`

```

1371 sh/.store in=\mmz@shname,
1372 sh=memoize-extract.\jobname.sh,
1373 bat/.store in=\mmz@batname,
1374 bat=memoize-extract.\jobname.bat,

```

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

```

1375 record/sh/begin/.code={%
1376   \newwrite\mmz@shout
1377   \immediate\mmz@openout\mmz@shout{\mmz@shname}%
1378 },
1379 record/sh/new extern/.code={%
1380   \begingroup

```

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

```

1381   \immediate\write\mmz@shout{\mmz@tex@extraction@systemcall}%
1382   \endgroup
1383 },
1384 record/sh/end/.code={%
1385   \immediate\closeout\mmz@shout
1386 },

```

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

```
1387 record/bat/begin/.code={%
1388   \newwrite\mmz@batout
1389   \immediate\mmz@openout\mmz@batout{\mmz@batname}%
1390 },
1391 record/bat/new extern/.code={%
1392   \begingroup
1393   \immediate\write\mmz@batout{\mmz@tex@extraction@systemcall}%
1394   \endgroup
1395 },
1396 record/bat/end/.code={%
1397   \immediate\closeout\mmz@batout
1398 },
```

### 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.

```
1399 makefile/.store in=\mmz@makefilename,
1400 makefile=memoize-extract.\jobname.makefile,
1401 }
```

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

```
1402 \begingroup
1403 \catcode\^^I=12
1404 \gdef\mmz@makefile@recipe@prefix{^^I}%
1405 \endgroup
```

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

```
1406 \mmzset{
1407   record/makefile/begin/.code={%
```

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

```
1408   \newwrite\mmz@makefileout
1409   \newtoks\mmz@makefile@externs
1410   \immediate\mmz@openout\mmz@makefileout{\mmz@makefilename}%
1411   \immediate\write\mmz@makefileout{.DEFAULT_GOAL = externs}%
1412   \immediate\write\mmz@makefileout{.PHONY: externs}%
1413 },
```

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`.

```
1414   record/makefile/new extern/.code={%
```

The target extern file:

```
1415   \immediate\write\mmz@makefileout{#1:}%
1416   \begingroup
```

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

```
1417   \immediate\write\mmz@makefileout{%
1418     \mmz@makefile@recipe@prefix\mmz@tex@extraction@systemcall}%
1419   \endgroup
```

Append the extern file to list of targets.

```
1420 \xtoksapp\mmz@makefile@externs{#1\space}%
1421 },
1422 record/makefile/end/.code={%
```

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

```
1423 \immediate\write\mmz@makefileout{externs: \the\mmz@makefile@externs}%
1424 \immediate\closeout\mmz@makefileout
1425 },
1426 }
```

### 4.3 T<sub>E</sub>X-based extraction

`extract/tex` We trigger the T<sub>E</sub>X-based extraction by inputting the `.mmz` record file.

```
1427 \mmzset{
1428   extract/tex/.code={%
1429     \begingroup
1430     \@input{\jobname.mmz}%
1431     \endgroup
1432   },
1433 }
```

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

```
\mmzUsedCCMemo 1434 \def\mmzUsedCMemo#1{}
\mmzUsedExtern 1435 \def\mmzUsedCCMemo#1{}
\mmzNewCMemo    1436 \def\mmzUsedExtern#1{}
\mmzNewCCMemo   1437 \def\mmzNewCMemo#1{}
\mmzPrefix     1438 \def\mmzNewCCMemo#1{}
               1439 \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<sub>E</sub>X 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`.

```
1440 \def\mmzNewExtern#1{%
```

The T<sub>E</sub>X executable expects the basename as the argument, so we strip away the `.pdf` suffix.

```
1441 \mmz@new@extern@i#1\mmz@temp
1442 }
1443 \def\mmz@new@extern@i#1.pdf\mmz@temp#2#3#4{%
1444 \begingroup
```

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

```
1445 \def\externbasepath{#1}%
1446 \def\pagenumber{#2}%
1447 \def\expectedwidth{#3}%
1448 \def\expectedheight{#4}%
```

Empty out the extraction log.

```
1449 \mmz@clear@extraction@log
```

Extract.

```
1450 \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.

```

1451 \let\mmz@extraction@error\mmz@pageextraction@error
1452 \mmz@check@extraction@log{tex}%
1453 \endgroup
1454 }

1455 \def\mmz@pageextraction@error{%
1456 \PackageError{memoize}{Extraction of extern page \pagenumber\space from
1457 document "jobname.pdf" using method "\extractionmethod" was
1458 unsuccessful.}{Check the log file to see if the extraction script was
1459 executed at all, and if it finished successfully. You might also want to
1460 inspect "\externbasepath.log", the log file of the embedded TeX compilation
1461 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;

```

1462 \def\mmz@tex@extraction@systemcall{%
1463 \mmzvalueof{tex extraction command}\space
1464 \mmzvalueof{tex extraction options}\space
1465 "\mmzvalueof{tex extraction script}"%
1466 }

```

**The default** system call for  $\TeX$ -based extern extraction. As this method, despite being  $\TeX$ -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 pdf $\TeX$ .<sup>3</sup> We perform the extraction by a little  $\TeX$  script, `memoize-extract-one`, inputted at the end of the value given to `tex extraction script`.

```

1467 \mmzset{
1468 tex extraction command/.initial=pdf $\TeX$ ,
1469 tex extraction options/.initial={%
1470 -halt-on-error
1471 -interaction=batchmode
1472 -jobname "\externbasepath"
1473 },
1474 tex extraction script/.initial={%
1475 \def\noexpand\fromdocument{\jobname.pdf}%
1476 \def\noexpand\pagenumber{\pagenumber}%
1477 \def\noexpand\expectedwidth{\expectedwidth}%
1478 \def\noexpand\expectedheight{\expectedheight}%
1479 \def\noexpand\logfile{\jobname.mmz.log}%
1480 \unexpanded{%
1481 \def\warningtemplate{%
1482 (latex) \noexpand\PackageWarning{memoize}{\warningtext}%
1483 (plain) \warning{memoize: \warningtext}%
1484 (context) \warning{memoize: \warningtext}%
1485 }}%
1486 \ifdef\XeTeXversion{}{%
1487 \def\noexpand\mmzpdfmajorversion{\the\pdfmajorversion}%
1488 \def\noexpand\mmzpdfminorversion{\the\pdfminorversion}%
1489 }%
1490 \noexpand\input memoize-extract-one

```

<sup>3</sup>I implemented the first version of  $\TeX$ -based extraction using L<sup>A</sup> $\TeX$  and package `graphicx`, and it was (running with pdf $\TeX$  engine) almost four times slower than the current plain  $\TeX$  implementation.

```

1491   },
1492 }
1493 </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.

```

1494 <{*extract-one}>
1495 \catcode`\@11\relax
1496 \def\@firstoftwo#1#2{#1}
1497 \def\@secondoftwo#1#2{#2}

```

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

```

1498 \ifdefined\XeTeXversion
1499 \else
1500   \ifdefined\luatexversion
1501     \def\pdfmajorversion{\pdfvariable majorversion}%
1502     \def\pdfminorversion{\pdfvariable minorversion}%
1503   \fi
1504   \ifdefined\mmzpdfmajorversion
1505     \pdfmajorversion\mmzpdfmajorversion\relax
1506   \fi
1507   \ifdefined\mmzpdfminorversion
1508     \pdfminorversion\mmzpdfminorversion\relax
1509   \fi
1510 \fi

```

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

```

1511 \newwrite\extractionlog

```

Are we requested to produce a log file?

```

1512 \ifdefined\logfile
1513   \immediate\openout\extractionlog{\logfile}%

```

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

```

1514   \def\doublewarning#1{%
1515     \message{#1}%
1516     \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.

```

1517     \immediate\write\extractionlog{%
1518       \ifdefined\warningtemplate\warningtemplate\else\warningtext\fi
1519     }%
1520   }%
1521 \else
1522   \let\doublewarning\message
1523 \fi
1524 \newif\ifforce
1525 \ifdefined\force
1526   \csname force\force\endcsname
1527 \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<sub>Y</sub>T<sub>E</sub>X.)

```

1528 </extract-one>
1529 <*mmz,extract-one>
1530 \def\mmz@tolerance{0.01pt}
1531 \def\mmz@if@roughly@equal#1#2#3{%
1532   \dimen0=\dimexpr#2-#3\relax
1533   \ifdim\dimen0<0pt
1534     \dimen0=-\dimen0\relax
1535   \fi
1536   \ifdim\dimen0>#1\relax
1537     \expandafter\@secondoftwo
1538   \else

```

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

```

1539     \expandafter\@firstoftwo
1540   \fi
1541 }%
1542 </mmz,extract-one>
1543 <*extract-one>

```

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

```

1544 \ifdefined\XeTeXversion
1545   \setbox0=\hbox{\XeTeXpdffile \fromdocument\space page \pagenumber media}%
1546 \else
1547   \ifdefined\luatexversion
1548     \saveimageresource page \pagenumber mediabox {\fromdocument}%
1549     \setbox0=\hbox{\useimageresource\lastsavedimageresourceindex}%
1550   \else
1551     \pdfximage page \pagenumber mediabox {\fromdocument}%
1552     \setbox0=\hbox{\pdfrefximage\pdflastximage}%
1553   \fi
1554 \fi

```

Check whether the extern page is of the expected size.

```

1555 \newif\ifbaddimensions
1556 \ifdefined\expectedwidth
1557   \ifdefined\expectedheight
1558     \mmz@if@roughly@equal{\mmz@tolerance}{\wd0}{\expectedwidth}{%
1559       \mmz@if@roughly@equal{\mmz@tolerance}{\ht0}{\expectedheight}%
1560     }%
1561     {\baddimensionstrue}%
1562   }{\baddimensionstrue}%
1563 \fi
1564 \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.

```

1565 \ifdefined\luatexversion
1566   \let\pdfpagewidth\pagewidth
1567   \let\pdfpageheight\pageheight
1568   \def\pdfhorigin{\pdfvariable horigin}%
1569   \def\pdfvorigin{\pdfvariable vorigin}%
1570 \fi
1571 \def\do@shipout{%
1572   \pdfpagewidth=\wd0
1573   \pdfpageheight=\ht0
1574   \ifdefined\XeTeXversion

```

```

1575     \hoffset -1 true in
1576     \voffset -1 true in
1577   \else
1578     \pdfhorigin=0pt
1579     \pdfvorigin=0pt
1580   \fi
1581   \shipout\box0
1582 }
1583 \ifbaddimensions
1584   \doublewarning{I refuse to extract page \pagenumber\space from
1585     "\fromdocument", because its size (\the\wd0 \space x \the\ht0) is not
1586     what I expected (\expectedwidth\space x \expectedheight)}%
1587   \ifforce\do@shipout\fi
1588 \else
1589   \do@shipout
1590 \fi

```

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

```

1591 \ifdefined\logfile
1592   \immediate\write\extractionlog{\noexpand\endinput}%
1593   \immediate\closeout\extractionlog
1594 \fi
1595 \bye
1596 </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.

```

1597 <*mmz>
1598 \mmzset{
1599   .install advice={setup key=auto, activation=deferred},

```

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

```

1600   begindocument/before/.append style={activation=immediate},
1601 }

```

**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  $\text{\LaTeX}$ , we will use the latest possible hook, `begindocument/end`, as we want to hack into commands defined by other packages. (The  $\text{\TeX}$  conditional needs to be defined before using it in `.append` code below.

```

1602 \newif\ifmmz@manual
1603 \mmzset{
1604   manual/.is if=mmz@manual,
1605   begindocument/end/.append code={%
1606     \ifmmz@manual
1607     \else
1608       \pgfkeysalso{activate deferred,activate deferred/.code={}}%
1609     \fi
1610   },

```

**Announce** Memoize's run conditions and handlers.

```

1611   auto/.cd,
1612   run if memoization is possible/.style={
1613     run conditions=\mmz@auto@rc@if@memoization@possible

```



```

1614 },
1615 run if memoizing/.style={run conditions=\mmz@auto@rc@if@memoizing},
1616 apply options/.style={
1617   bailout handler=\mmz@auto@bailout,
1618   outer handler=\mmz@auto@outer,
1619 },
1620 memoize/.style={
1621   run if memoization is possible,
1622   apply options,
1623   inner handler=\mmz@auto@memoize
1624 },
1625 *latex
1626 noop/.style={run if memoization is possible, noop \AdviceType},
1627 noop command/.style={apply options, inner handler=\mmz@auto@noop},
1628 noop environment/.style={
1629   outer handler=\mmz@auto@noop@env, bailout handler=\mmz@auto@bailout},
1630 /latex
1631 plain, context noop/.style={inner handler=\mmz@auto@noop},
1632 nomemoize/.style={noop, options=disable},
1633 replicate/.style={run if memoizing, inner handler=\mmz@auto@replicate},
1634 to context/.style={run if memoizing, outer handler=\mmz@auto@tocontext},
1635 }

```

**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.)

```

1636 \mmzset{
1637   auto/abort/.style={run conditions=\mmzAbort},
1638 }

```

And the same for unmemoizable:

```

1639 \mmzset{
1640   auto/unmemoizable/.style={run conditions=\mmzUnmemoizable},
1641 }

```

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.

```

1642 \ifdef\luatexversion{%
1643   \mmzset{auto=\savepos{abort}}
1644 }{%
1645   \mmzset{
1646     auto=\pdfsavepos{abort},
1647     auto=\errmessage{abort},
1648   }
1649 }

```

`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.

```

1650 \def\mmz@auto@rc@if@memoization@possible{%
1651   \ifmemoize
1652     \ifinmemoize
1653     \else
1654       \AdviceRuntrue
1655     \fi
1656   \fi
1657 }

```

`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).

```
1658 \def\mmz@auto@rc@if@memoizing{%
1659   \ifmemoizing\AdviceRuntrue\fi
1660 }
```

`\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`.

```
1661 </mmz>
1662 <nommz> \def\mmznext#1{\ignorespaces}
1663 <*mmz>
1664 \def\mmznext#1{\gdef\mmz@next{#1}\ignorespaces}
1665 \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.

```
1666 \def\mmz@auto@outer{%
1667   \begingroup
1668   \mmzAutoInit
1669   \AdviceCollector
1670 }
1671 \def\mmz@auto@bailout{%
1672   \mmznext{}%
1673 }
```

`\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.

```
1674 \def\mmzAutoInit{%
1675   \ifdefempty\AdviceOptions{\expandafter\mmzset\expandafter{\AdviceOptions}}%
1676   \ifdefempty\mmz@next{\expandafter\mmzset\expandafter{\mmz@next}\mmznext{}}%
1677   \eappto\AdviceRawCollectorOptions{\expandonce\mmzRawCollectorOptions}%
1678 }
```

`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 `TeX` 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.)

```

1679 \long\def\mmz@auto@memoize#1{%
1680   \expanded{%
1681     \noexpand\Memoize
1682     {\expandonce\AdviceReplaced\unexpanded{#1}}%
1683     {\expandonce\AdviceOriginal\unexpanded{#1}}%
1684     \ifmmz@ignorespaces\ignorespaces\fi
1685   }%
1686 }

```

`noop` The no-operation handler can be used to apply certain options for the span of the execution `\mmz@auto@noop` 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<sup>A</sup>T<sub>E</sub>X 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.

```

1687 \long\def\mmz@auto@noop#1{%
1688   \expandafter\mmz@maybe@scantokens\expandafter{\AdviceOriginal#1}%
1689   \expandafter\endgroup
1690   \ifmmz@ignorespaces\ignorespaces\fi
1691 }

```

In L<sup>A</sup>T<sub>E</sub>X, and only there, commands and environments need separate treatment. As L<sup>A</sup>T<sub>E</sub>X 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.

```

1692 <*\latex>
1693 \def\mmz@auto@noop@env{%
1694   \AddToHookNext{env/\AdviceName/begin}{%
1695     \mmzAutoInit
1696     \ifmmz@ignorespaces\ignorespacesafterend\fi
1697   }%
1698   \AdviceOriginal
1699 }
1700 </\latex>

```

`replicate` This inner handler writes a copy of the handled command or environment's invocation into `\mmz@auto@replicate` 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.

```

1701 \def\mmz@auto@replicate#1{%
1702   \begingroup
1703   \let\mmz@auto@replicate@expansion\unexpanded
1704   \expandafter\pgfqkeys\expanded{{\mmz/auto/replicate}{\AdviceOptions}}%
1705 <\latex> \let\protect\noexpand
1706   \expanded{%
1707     \endgroup
1708     \noexpand\gtoksapp\noexpand\mmzCCMemo{%
1709       \expandonce\AdviceReplaced\mmz@auto@replicate@expansion{#1}}%
1710     \expandonce\AdviceOriginal\unexpanded{#1}%

```

```

1711 }%
1712 }
1713 \pgfqkeys{/mmz/auto/replicate}{
1714   expanded/.code={\let\mmz@auto@replicate@expansion\@firstofone},
1715 }

```

**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 `\AdviceGetOriginalCsname`.

```

1716 \def\mmz@auto@tocontext{%
1717   \expanded{%
1718     \noexpand\pgfkeysvalueof{/mmz/context/.@cmd}%
1719     original "\AdviceNamespace" csname "\AdviceCsname"={%
1720       \noexpand\expanded{%
1721         \noexpand\noexpand\noexpand\meaning
1722         \noexpand\AdviceCsnameGetOriginal{\AdviceNamespace}{\AdviceCsname}%
1723       }%
1724     }%
1725   }%
1726   \pgfeov
1727   \AdviceOriginal
1728 }

```

## 5.1 L<sup>A</sup>T<sub>E</sub>X-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.

```

1729 (*latex)
1730 \mmzset{
1731   auto/.cd,
1732   ref/.style={outer handler=\mmz@auto@ref\mmzNoRef, run if memoizing},
1733   force ref/.style={outer handler=\mmz@auto@ref\mmzForceNoRef, run if memoizing},
1734 }
1735 \mmzset{
1736   auto=\ref{ref},
1737   auto=\pageref{ref},
1738   auto=\label{run if memoizing, outer handler=\mmz@auto@label},
1739   auto=\index{replicate, args=m, expanded},
1740 }

```

**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.

```

1741 \def\mmz@auto@ref#1#2#{\mmz@auto@ref@i#1{#2}}
1742 \def\mmz@auto@ref@i#1#2#3{%
1743   #1{#3}%
1744   \AdviceOriginal#2{#3}%
1745 }

```

`\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.

```

1746 \def\mmzForceNoRef#1{%
1747   \mmz@mtoc@csname{r@#1}%
1748   \ignorespaces
1749 }
1750 \def\mmzNoRef#1{%
1751   \ifcsundef{r@#1}{\mmzAbort}{\mmzForceNoRef{#1}}%
1752   \ignorespaces
1753 }

```

`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
1754 \mmzset{
1755   auto/.cd,
1756   refrange/.style={outer handler=\mmz@auto@refrange\mmzNoRef,
1757     bailout handler=\relax, run if memoizing},
1758   force refrange/.style={outer handler=\mmz@auto@refrange\mmzForceNoRef,
1759     bailout handler=\relax, run if memoizing},
1760 }

1761 \def\mmz@auto@refrange#1#2#{\mmz@auto@refrange@i#1{#2}}
1762 \def\mmz@auto@refrange@i#1#2#3#4{%
1763   #1{#3}%
1764   #1{#4}%
1765   \AdviceOriginal#2{#3}{#4}%
1766 }

```

`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`.

```

1767 \mmzset{
1768   auto/.cd,
1769   multiref/.style={outer handler=\mmz@auto@multiref\mmzNoRef,
1770     bailout handler=\relax, run if memoizing},
1771   force multiref/.style={outer handler=\mmz@auto@multiref\mmzForceNoRef,
1772     bailout handler=\relax, run if memoizing},
1773 }
1774 \def\mmz@auto@multiref#1#2#{\mmz@auto@multiref@i#1{#2}}
1775 \def\mmz@auto@multiref@i#1#2#3{%
1776   \forcsvlist{#1}{#3}%
1777   \AdviceOriginal#2{#3}%
1778 }

```

`\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`.

```

1779 \def\mmz@auto@label#1{%
1780   \xtoksapp\mmzCCMemo{%
1781     \noexpand\mmzLabel{#1}{\expandonce\@currentlabel}%
1782   }%
1783   \AdviceOriginal{#1}%
1784 }

```

`\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.

```

1785 \def\mmzLabel#1#2{%
1786   \begingroup
1787   \def\@currentlabel{#2}%
1788   \label{#1}%
1789   \endgroup
1790 }
1791 </latex>

```

## 6 Support for various classes and packages

```

1792 <*latex>
1793 \AddToHook{shipout/before}[memoize]{\global\advance\mmzExtraPages-1\relax}
1794 \AddToHook{shipout/after}[memoize]{\global\advance\mmzExtraPages1\relax}
1795 \mmzset{auto=\DiscardShipoutBox{
1796   outer handler=\global\advance\mmzExtraPages1\relax\AdviceOriginal}}
1797 </latex>

```

### 6.1 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.

```

1798 \mmzset{
1799   begindocument/before/.append code={%
1800 <latex>   \@ifpackageloaded{tikz}{%
1801 <plain, context>   \ifdefined\tikz
1802   \input advice-tikz.code.tex
1803 <latex>   }{)%
1804 <plain, context>   \fi

```

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

```

1805   \mmzset{%
1806     auto/memoize tikz/.style={
1807       memoize,
1808       at begin memoization=\edef\mmz@pgfpictureid{%
1809         \the\pgf@picture@serial@count
1810       },
1811       at end memoization=\xtoksapp\mmzCCMemo{%
1812         \unexpanded{%
1813           \global\expandafter\advance\csname pgf@picture@serial@count\endcsname
1814         }%
1815         \the\numexpr\pgf@picture@serial@count-\mmz@pgfpictureid\relax\relax
1816       },
1817     },
1818     auto=\tikz{memoize tikz, collector=\AdviceCollectTikZArguments},
1819     auto={tikzpicture}{memoize tikz},
1820   }%
1821 },
1822 }

```

### 6.2 Forest

Forest will soon feature extensive memoization support, but for now, let's just enable the basic, single extern externalization.

```

1823 <*latex>
1824 \mmzset{
1825   begindocument/before/.append code={%
1826     \@ifpackageloaded{forest}{%
1827       \mmzset{
1828         auto={forest}{memoize},

```

Yes, `\Forest` is defined using `xparse`.

```
1829     auto=\Forest{memoize},
1830   }%
1831 }{}%
1832 },
1833 }
1834 </latex>
```

### 6.3 Beamer

The Beamer code is explained in [M§4.2.4](#).

```
1835 <*latex>
1836 \AddToHook{begindocument/before}{\@ifclassloaded{beamer}{%
1837   \mmzset{per overlay/.style={
1838     /mmz/context={%
1839       overlay=\csname beamer@overlaynumber\endcsname,
1840       pauses=\ifmemoizing
1841         \mmzBeamerPauses
1842       \else
1843         \expandafter\the\csname c@beamerpauses\endcsname
1844       \fi
1845     },
1846     /mmz/at begin memoization={%
1847       \xdef\mmzBeamerPauses{\the\c@beamerpauses}%
1848       \xtoksapp\mmzCMemo{%
1849         \noexpand\mmzSetBeamerOverlays{\mmzBeamerPauses}{\beamer@overlaynumber}}%
1850       \gtoksapp\mmzCCMemo{%
1851         \only<\mmzBeamerOverlays>{}}%
1852     },
1853     /mmz/at end memoization={%
1854       \xtoksapp\mmzCCMemo{%
1855         \noexpand\setcounter{beamerpauses}{\the\c@beamerpauses}}%
1856     },
1857     /mmz/per overlay/.code={},
1858   }}
1859 \def\mmzSetBeamerOverlays#1#2{%
1860   \ifnum\c@beamerpauses=#1\relax
1861     \gdef\mmzBeamerOverlays{#2}%
1862     \ifnum\beamer@overlaynumber<#2\relax \mmz@temptrue \else \mmz@tempfalse \fi
1863   \else
1864     \mmz@temptrue
1865   \fi
1866   \ifmmz@temp
1867     \appto\mmzAtBeginMemoization{%
1868       \gtoksapp\mmzCMemo{\mmzSetBeamerOverlays{#1}{#2}}}%
1869   \fi
1870 }%
1871 }{}%
1872 </latex>
```

### 6.4 Morewrites

Use the old grammar for `\openin` and `\openout` as a temporary workaround. `prefixes` containing spaces must be quoted manually.

```
1873 <*latex>
1874 \AddToHook{begindocument/before}{%
1875   \@ifpackageloaded{morewrites}{%
1876     \def\mmz@openin#1#2{\openin#1=#2\relax}%
1877     \def\mmz@openout#1#2{\openout#1=#2\relax}%
1878   }{}%
```

```
1879 }
1880 </latex>
```

## 6.5 Biblatex

```
1881 <*latex>
1882 \mmzset{
1883   beginindocument/before/.append style={%
1884     auto=\blx@bbl@entry{outer handler=\mmz@biblatex@entry},
1885     auto/cite/.style={run if memoizing, outer handler=\mmz@biblatex@cite},
1886     auto/cites/.style={run if memoizing, outer handler=\mmz@biblatex@cites},
1887     auto=\cite{cite},
1888     auto=\cites{cites},
1889   }%
1890 }
```

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

```
1891 \def\mmz@biblatex@entry#1#2\endentry{%
1892   \protected@edef\mmz@temp{\pdf@mdfivesum{#2}}%
1893   \global\cslet{mmz@bbl@#1}\mmz@temp
1894   \AdviceOriginal{#1}#2\endentry
1895 }
```

`\mmz@biblatex@cite` This macro puts the cites reference keys into the context, and adds the handled `\cite` command to the cc-memo.

```
1896 \def\mmz@biblatex@cite#1#{\mmz@biblatex@cite@i{#1}}
1897 \def\mmz@biblatex@cite@i#1#2{%
1898   \forcsvlist\mmz@biblatex@cite@do@key{#2}%
1899   \xtoksapp\mmzCCMemo{%
1900     \noexpand\setbox0\noexpand\hbox{%
1901       \expandonce\AdviceOriginal\unexpanded{#1}{#2}%
1902     }}%
1903   \AdviceOriginal#1{#2}%
1904 }
1905 \def\mmz@biblatex@cite@do@key#1{%
1906   \mmz@mtoc@csname{mmz@bbl@#1}%
1907   \ifcsdef{mmz@bbl@#1}{-}{\mmzAbort}%
1908 }
```

`\mmz@biblatex@cites` This macro puts the cites reference keys into the context, and adds the handled `\cites` command to the cc-memo.

```
1909 \def\mmz@biblatex@cites{%
1910   \mmz@temptoks{}%
1911   \mmz@biblatex@cites@i
1912 }
1913 \def\mmz@biblatex@cites@i{%
1914   \futurelet\mmz@temp\mmz@biblatex@cites@ii
1915 }
1916 \def\mmz@biblatex@cites@ii{%
1917   \mmz@tempfalse
1918   \ifx\mmz@temp\bgroup
1919     \mmz@temptrue
1920   \else
1921     \ifx\mmz@temp[%]
1922       \mmz@temptrue
1923     \fi
1924   \fi
1925   \ifmmz@temp
1926     \expandafter\mmz@biblatex@cites@iii
1927   \else
```



```

1928 \expandafter\mmz@biblatex@cites@z
1929 \fi
1930 }
1931 \def\mmz@biblatex@cites@iii#1#2{\mmz@biblatex@cites@iv{#1}}
1932 \def\mmz@biblatex@cites@iv#1#2{%
1933 \forcsvlist\mmz@biblatex@cite@do@key{#2}%
1934 \toksapp\mmz@temptoks{#1{#2}}%
1935 \mmz@biblatex@cites@i
1936 }
1937 \def\mmz@biblatex@cites@z{%
1938 \xtoksapp\mmzCCMemo{%
1939 \noexpand\setbox0\noexpand\hbox{%
1940 \expandonce\AdviceOriginal\the\mmz@temptoks
1941 }}%
1942 \expandafter\AdviceOriginal\the\mmz@temptoks
1943 }
1944 </latex>

```

## 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 `begindocument/end` 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  $\text{\LaTeX}$ , we can automatically execute these hooks at appropriate places:

```

1945 <*latex>
1946 \AddToHook{begindocument/before}{\mmzset{begindocument/before}}
1947 \AddToHook{begindocument}{\mmzset{begindocument}}
1948 \AddToHook{begindocument/end}{\mmzset{begindocument/end}}
1949 \AddToHook{enddocument/afterlastpage}{\mmzset{enddocument/afterlastpage}}
1950 </latex>

```

In plain  $\text{\TeX}$ , 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  $\text{\ConTeXt}$  as well, as I'm not sure where to execute them — please help!

```

1951 <*plain, context>
1952 \mmzset{
1953 begin document/.style={begindocument/before, begindocument, begindocument/end},
1954 end document/.style={enddocument/afterlastpage},
1955 }
1956 </plain, context>

```

Formats other than plain  $\text{\TeX}$  need a way to prevent extraction during package-loading.

```

1957 <!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  $\text{\M}$ §2.6).

```

1958 </mmz>
1959 <mmz, nommz>\InputIfFileExists{memoize.cfg}{}{}
1960 <*mmz>

```

For formats other than plain  $\text{\TeX}$ , 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.

```
1961 (!plain) \let\mmz@initial@extraction@method\mmz@extraction@method
```

**Process** the package options (except in plain T<sub>E</sub>X).

```
1962 (*latex)
1963 \DeclareUnknownKeyHandler[mmz]{%
1964   \expanded{\noexpand\pgfqkeys{/mmz}{#1\IfBlankF{#2}{-={#2}}}}
1965 \ProcessKeyOptions[mmz]
1966 (/latex)
1967 (context) \expandafter\mmzset\expandafter{\currentmoduleparameters}
```

In L<sup>A</sup>T<sub>E</sub>X, nomemoize has to process package options as well, otherwise L<sup>A</sup>T<sub>E</sub>X will complain.

```
1968 (/mmz)
1969 (*latex & nommz)
1970 \DeclareUnknownKeyHandler[mmz]{}
1971 \ProcessKeyOptions[mmz]
1972 (/latex & nommz)
```

**Extern extraction** We redefine `extract` to immediately trigger extraction. This is crucial in plain T<sub>E</sub>X, where extraction must be invoked after loading the package, but also potentially useful in other formats when package options include `extract=no`.

```
1973 (*mmz)
1974 \mmzset{
1975   extract/.is choice,
1976   extract/.default=\mmz@extraction@method,
```

But only once:

```
1977   extract/.append style={
1978     extract/.code={\PackageError{memoize}{Key "extract" was invoked twice.}{In
1979       principle, externs should be extracted only once. If you really want
1980       to extract again, execute "extract/<method>".}},
1981   },
```

In formats other than plain T<sub>E</sub>X, we remember the current `extract` code and then trigger the extraction.

```
1982 (!plain) /utils/exec={\pgfkeysgetvalue{/mmz/extract/.@cmd}\mmz@temp@extract},
1983 (!plain) extract=\mmz@extraction@method,
1984 }
```

Option `extract=no` (which only exists in formats other than plain T<sub>E</sub>X) should allow for an explicit invocation of `extract` in the preamble.

```
1985 (*!plain)
1986 \def\mmz@temp{no}
1987 \ifx\mmz@extraction@method\mmz@temp
1988   \pgfkeyslet{/mmz/extract/.@cmd}\mmz@temp@extract
1989   \let\mmz@extraction@method\mmz@initial@extraction@method
1990 \fi
1991 \let\mmz@temp@extract\relax
1992 (/!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.

```
1993 \ifnum\pdf@draftmode=1
1994   \PackageWarning{memoize}{No externalization will be performed in the draft mode}%
1995 \fi
1996 (/mmz)
```

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

```
1997 (*nommz, mmzable & generic)
1998 \pgfkeys{%
1999   /handlers/.meaning to context/.code={},
2000   /handlers/.value to context/.code={},
2001 }
2002 \let\mmzAbort\relax
2003 \let\mmzUnmemoizable\relax
2004 \newcommand\IfMemoizing[2] [] {\@secondoftwo}
2005 \let\mmzNoRef\@gobble
2006 \let\mmzForceNoRef\@gobble
2007 \newtoks\mmzContext
2008 \newtoks\mmzContextExtra
2009 \newtoks\mmzCMemo
2010 \newtoks\mmzCCMemo
2011 \newcount\mmzExternPages
2012 \newcount\mmzExtraPages
2013 \let\mmzTracingOn\relax
2014 \let\mmzTracingOff\relax
2015 (/nommz, mmzable & generic)
```

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

```
2016 (*mmz, nommz, mmzable)
2017 (plain) \resetatcatcode
2018 (context) \stopmodule
2019 (context) \protect
2020 (/mmz, nommz, mmzable)
```

That's all, folks!

## 8 Auxiliary packages

### 8.1 Extending commands and environments with Advice

```
2021 (*main)
2022 (latex) \ProvidesPackage{advice}[2024/01/02 v1.1.0 Extend commands and environments]
2023 (context) %D \module[
2024 (context) %D     file=t-advice.tex,
2025 (context) %D     version=1.1.0,
2026 (context) %D     title=Advice,
2027 (context) %D     subtitle=Extend commands and environments,
2028 (context) %D     author=Saso Zivanovic,
2029 (context) %D     date=2024-01-02,
2030 (context) %D     copyright=Saso Zivanovic,
2031 (context) %D     license=LPL,
2032 (context) %D ]
2033 (context) \writestatus{loading}{ConTeXt User Module / advice}
2034 (context) \unprotect
2035 (context) \startmodule[advice]
```

#### Required packages

```
2036 (plain, context) \input miniltx
2037 (latex) \RequirePackage{collargs}
2038 (plain) \input collargs
2039 (context) \input t-collargs
```

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

```
2040 (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.

```

2041 \pgfkeys{
2042   /handlers/.install advice/.code={%
2043     \edef\auto@install@namespace{\pgfkeyscurrentpath}%
2044     \def\advice@install@setupkey{advice}%
2045     \def\advice@install@activation{immediate}%
2046     \pgfqkeys{/advice/install}{#1}%
2047     \expanded{\noexpand\advice@install
2048       {\auto@install@namespace}%
2049       {\advice@install@setupkey}%
2050       {\advice@install@activation}}%
2051   }%
2052 },

```

**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`.

```

2053 /advice/install/.cd,
2054 setup key/.store in=\advice@install@setupkey,
2055 activation/.is choice,
2056 activation/.append code=\def\advice@install@activation{#1},
2057 activation/immediate/.code={},
2058 activation/deferred/.code={},
2059 }

```

`#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.

```

2060 \def\advice@install#1#2#3{%
    Switch to the installation keypath.
2061 \pgfqkeys{#1}{%

```

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

```

auto key'
auto csname' 2062 #2/.code 2 args={%

```

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

```

2063 \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.)

```

2064 \pgfqkeys{#1}{try activate, activate={##1}}%
2065 },

```

A variant without activation.

```

2066 #2'/.code 2 args={%
2067 \AdviceSetup{#1}{#2}{##1}{\pgfqkeys{#1/#2}{##2}}%
2068 },
2069 #2 csname/.style 2 args={
2070 #2/.expand once=\expandafter{\csname ##1\endcsname}{##2},
2071 },
2072 #2 csname'/.style 2 args={
2073 #2'/.expand once=\expandafter{\csname ##1\endcsname}{##2},
2074 },

```

```

2075 #2 key/.style 2 args={
2076 #2/.expand once=%
2077 \expandafter{\csname pgfk###1/.@cmd\endcsname}%
2078 {collector=\advice@pgfkeys@collector,##2},
2079 },
2080 #2 key'/.style 2 args={
2081 #2'/.expand once=%
2082 \expandafter{\csname pgfk###1/.@cmd\endcsname}%
2083 {collector=\advice@pgfkeys@collector,##2},
2084 },

```

**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).

```

2085 activation/.style={/advice/activation/##1={#1}},
2086 activation=#3,

```

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

```

2087 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).

```

2088 activate csname/.style={activate/.expand once={\csname###1\endcsname}},
2089 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**

```

2090 activate key/.style={activate@key={#1/activate}{##1}},
2091 deactivate key/.style={activate@key={#1/deactivate}{##1}},
2092 activate@key/.code n args=2{%
2093 \def\advice@temp{%
2094 \def\advice@do####1{%
2095 \eappto\advice@temp{\expandonce{\csname pgfk#####1/.@cmd\endcsname}}}%
2096 \forcsvlist\advice@do{##2}%
2097 \pgfkeysalso{##1/.expand once=\advice@temp}%
2098 },

```

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

```

2099 #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 **bailout handler** a crucial role in the handler definitions, see section 8.1.3.

```

collector
  args
collector options
clear collector options
raw collector options
clear raw collector options
inner handler
  options
clear options
2100 run conditions/.store in=\AdviceRunConditions,
2101 bailout handler/.store in=\AdviceBailoutHandler,
2102 outer handler/.store in=\AdviceOuterHandler,
2103 collector/.store in=\AdviceCollector,
2104 collector options/.code={\appto\AdviceCollectorOptions{##1}},
2105 clear collector options/.code={\def\AdviceCollectorOptions{}},
2106 raw collector options/.code={\appto\AdviceRawCollectorOptions{##1}},
2107 clear raw collector options/.code={\def\AdviceRawCollectorOptions{}},
2108 args/.store in=\AdviceArgs,
2109 inner handler/.store in=\AdviceInnerHandler,
2110 options/.code={\appto\AdviceOptions{##1}},
2111 clear options/.code={\def\AdviceOptions{}},

```

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

```
2112 .unknown/.code={%
2113   \eappto{\AdviceOptions}{,\pgfkeyscurrentname={\unexpanded{##1}}}%
2114 },
```

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

```
2115 run conditions/.default=\AdviceRuntrue,
2116 bailout handler/.default=\relax,
2117 outer handler/.default=\advice@default@outer@handler,
2118 collector/.default=\advice@CollectArgumentsRaw,
2119 collector options/.value required,
2120 raw collector options/.value required,
2121 args/.default=\advice@noargs,
2122 inner handler/.default=\advice@error@noinnerhandler,
2123 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.

```
2124 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.

```
2125 after setup/.code={\appto\AdviceAfterSetup{##1}},
```

In  $\text{\LaTeX}$ , 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.

```
2126 <latex> #1/#2=\begin{run conditions=\advice@begin@rc},
2127 }%
2128 }
```

### 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.)

```
2129 \def\AdviceSetup#1#2#3{%
```

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

```
2130 \begingroup
2131 \def\AdviceName{#3}%
2132 \advice@def@AdviceCname
```

Command, complain, or environment?

```
2133 \collargs@cs@cases{#3}{%
2134 \def\AdviceType{command}%
2135 \advice@setup@init@command
2136 \advice@setup@i{#3}{#1}{#3}%
2137 }{%
2138 \advice@error@advice@notcs{#1/#2}{#3}%
2139 }{%
2140 \def\AdviceType{environment}%
2141 \advice@setup@init@environment
2142 <latex> \advice@setup@i{#3}%
2143 <plain> \expandafter\advice@setup@i\expandafter{\csname #3\endcsname}%
2144 <context> \expandafter\advice@setup@i\expandafter{\csname start#3\endcsname}%
2145 {#1}{#3}%
2146 }%
2147 }
```

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  $\LaTeX$  environments; when submitting environment `foo`, we get `#1=#3=foo`. But in plain  $\TeX$ , `#1=\foo` and `#3=foo`, and in  $\ConTeXt$ , `#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`.)

```
2148 \def\advice@setup@i#1#2#3#4{%
```

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

```
2149 \advice@setup@init@i{#2}{#1}%
2150 \advice@setup@init@I{#2}{#1}%
2151 \def\AdviceAfterSetup{}
```

Apply the setup code/keys.

```
2152 #4%
```

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

```
2153 \advice@setup@save{#2}{#1}%
2154 }
```

**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.

```
2155 \def\advice@setup@init@common{%
2156 \def\AdviceRunConditions{\AdviceRuntrue}%
2157 \def\AdviceBailoutHandler{\relax}%
2158 \def\AdviceOuterHandler{\advice@default@outer@handler}%
```

```

2159 \def\AdviceCollector{\advice@CollectArgumentsRaw}%
2160 \def\AdviceCollectorOptions{}%
2161 \def\AdviceInnerHandler{\advice@error@noinnerhandler}%
2162 \def\AdviceOptions{}%
2163 }
2164 \def\advice@setup@init@command{%
2165 \advice@setup@init@common
2166 \def\AdviceRawCollectorOptions{}%
2167 \def\AdviceArgs{\advice@noargs}%
2168 }
2169 \def\advice@setup@init@environment{%
2170 \advice@setup@init@common
2171 \edef\AdviceRawCollectorOptions{%
2172 \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.

```

2173 \noexpand\collargsEndTagtrue
2174 }%
2175 \def\AdviceArgs{+b}%
2176 }

```

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

```

2177 \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).

```

2178 \def\advice@init@i#1#2{\csname advice@i#1//\string#2\endcsname}
2179 \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.

```

2180 \let\advice@setup@init@i\advice@init@i
2181 \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.

```

2182 \def\advice@setup@save#1#2{%
2183 \expanded{%

```



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

```

2184 \endgroup
2185 \noexpand\csdef{advice@i#1//\string#2}{%
2186   \def\noexpand\AdviceRunConditions{\expandonce\AdviceRunConditions}%
2187   \def\noexpand\AdviceBailoutHandler{\expandonce\AdviceBailoutHandler}%
2188 }%
2189 \noexpand\csdef{advice@I#1//\string#2}{%
2190   \def\noexpand\AdviceOuterHandler{\expandonce\AdviceOuterHandler}%
2191   \def\noexpand\AdviceCollector{\expandonce\AdviceCollector}%
2192   \def\noexpand\AdviceRawCollectorOptions{%
2193     \expandonce\AdviceRawCollectorOptions}%
2194   \def\noexpand\AdviceCollectorOptions{\expandonce\AdviceCollectorOptions}%
2195   \def\noexpand\AdviceArgs{\expandonce\AdviceArgs}%
2196   \def\noexpand\AdviceInnerHandler{\expandonce\AdviceInnerHandler}%
2197   \def\noexpand\AdviceOptions{\expandonce\AdviceOptions}%
2198 }%
2199 \expandonce{\AdviceAfterSetup}%
2200 }%
2201 }

```

**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.)

```

2202 \pgfkeys{
2203   /advice/activation/deferred/.style={
2204     #1/activate/.style={%
2205       activate deferred/.append style={#1/activate={##1}}},
2206     #1/deactivate/.style={%
2207       activate deferred/.append style={#1/deactivate={##1}}},
2208     #1/force activate/.style={%
2209       activate deferred/.append style={#1/force activate={##1}}},
2210     #1/try activate/.style={%
2211       activate deferred/.append style={#1/try activate={##1}}},
2212   },

```

**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.

**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.)

```

2213   /advice/activation/immediate/.style={
2214     #1/activate/.code={%
2215       \forcsvlist{\advice@activate{#1}}{##1}%
2216       \advice@activate@forcefalse
2217       \advice@activate@tryfalse
2218     },
2219     #1/deactivate/.code={%
2220       \forcsvlist{\advice@deactivate{#1}}{##1}%
2221       \advice@activate@forcefalse
2222       \advice@activate@tryfalse
2223     },
2224     #1/force activate/.is if=advice@activate@force,
2225     #1/try activate/.is if=advice@activate@try,

```

```

2226 },
2227 }
2228 \newif\ifadvice@activate@force
2229 \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.

```

2230 \def\advice@original@csname#1#2{advice@o#1//\string#2}
2231 \def\advice@original@cs#1#2{\csname advice@o#1//\string#2\endcsname}
2232 \def\AdviceGetOriginal#1#2{%
2233   \ifcsname advice@o#1//\string#2\endcsname
2234     \expandonce{\csname advice@o#1//\string#2\expandafter\endcsname\expandafter}%
2235   \else
2236     \expandafter\noexpand\expandafter#2%
2237   \fi
2238 }

```

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

```

2239 \begingroup
2240 \catcode`\/=0
2241 \catcode`\=12
2242 /gdef/advice@backslash@other{\}%
2243 /endgroup
2244 \def\AdviceCsnameGetOriginal#1#2{%
2245   \ifcsname advice@o#1//\advice@backslash@other#2\endcsname
2246     \expandonce{\csname advice@o#1//\advice@backslash@other#2\expandafter\endcsname
2247       \expandafter}%
2248   \else
2249     \expandonce{\csname#2\expandafter\endcsname\expandafter}%
2250   \fi
2251 }

```

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

```

\advice@deactivate
2252 \def\advice@activate#1#2{%
2253   \collargs@cs@cases{#2}%
2254   {\advice@activate@cmd{#1}{#2}}%
2255   {\advice@error@activate@notcsorenv{#1}}%
2256   {\advice@activate@env{#1}{#2}}%
2257 }
2258 \def\advice@deactivate#1#2{%
2259   \collargs@cs@cases{#2}%
2260   {\advice@deactivate@cmd{#1}{#2}}%
2261   {\advice@error@activate@notcsorenv{de}{#1}}%

```

```

2262     {\advice@deactivate@env{#1}{#2}}%
2263 }

```

`\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.)

```

2264 \def\advice@activate@cmd#1#2{%

```

Is the command defined?

```

2265   \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).

```

2266     \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?

```

2267     \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.

```

2268         \advice@activate@error@activated{#1}{#2}{Command}{already}%
2269     }{%

```

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.

```

2270         \ifadvice@activate@force
2271         \advice@activate@cmd@do{#1}{#2}%
2272     \else
2273         \advice@activate@error@activated{#1}{#2}{Command}{already}%
2274     \fi
2275 }%
2276 }{%

```

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

```

2277     \advice@activate@cmd@do{#1}{#2}%
2278 }%
2279 }{%
2280     \advice@activate@error@undefined{#1}{#2}{Command}{}%
2281 }%
2282 }

```

`\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).

```

2283 \def\advice@deactivate@cmd#1#2{%
2284   \ifdef{#2}{%
2285     \ifcsdef{\advice@original@csname{#1}{#2}}{%
2286       \advice@if@our@definition{#1}{#2}{%
2287         \advice@deactivate@cmd@do{#1}{#2}%
2288       }{%
2289         \advice@deactivate@error@changed{#1}{#2}%
2290       }%
2291     }{%
2292       \advice@activate@error@activated{#1}{#2}{Command}{not yet}%
2293     }%
2294   }{%
2295     \advice@activate@error@undefined{#1}{#2}{Command}{de}%
2296   }%
2297 }

```

`\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).

```

2298 \def\advice@if@our@definition#1#2{%
2299   \protected\def\advice@temp{\advice@handle{#1}{#2}}%
2300   \ifx#2\advice@temp
2301     \expandafter\@firstoftwo
2302   \else
2303     \expandafter\@secondoftwo
2304   \fi
2305 }

```

`\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  $\varepsilon$ -TEX anyway, we don’t have to pay attention to L<sup>A</sup>T<sub>E</sub>X’s robust commands by redefining their “inner” command. Protecting our replacement suffices.)

```

2306 \def\advice@activate@cmd@do#1#2{%
2307   \cslet{\advice@original@csname{#1}{#2}}#2%
2308   \protected\def#2{\advice@handle{#1}{#2}}%
2309   \PackageInfo{advice (#1)}{Activated command "\string#2"}%
2310 }

```

`\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.

```

2311 \def\advice@deactivate@cmd@do#1#2{%
2312   \letcs#2{\advice@original@csname{#1}{#2}}%
2313   \csundef{\advice@original@csname{#1}{#2}}%
2314   \PackageInfo{advice (#1)}{Deactivated command "\string#2"}%
2315 }

```

### 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<sup>A</sup>T<sub>E</sub>X’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 nor 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`.

```
2316 \def\advice@handle#1#2{%
2317   \advice@init@i{#1}{#2}%
2318   \AdviceRunfalse
2319   \AdviceRunConditions
2320   \advice@handle@rc{#1}{#2}%
2321 }
```

`\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.

```
2322 \def\advice@handle@rc#1#2{%
2323   \ifAdviceRun
2324     \expandafter\advice@handle@outer
2325   \else
```

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

```
2326     \AdviceBailoutHandler
2327     \expandafter\advice@original@cs
2328   \fi
2329   {#1}{#2}%
2330 }
```

`\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 `LATEX`, `\foo` in plain `TEX` and `\startfoo` in `ConTEXt`.
- `\AdviceOriginal` executes the original definition of the handled command or environment.

```
2331 \def\advice@handle@outer#1#2{%
2332   \def\AdviceNamespace{#1}%
2333   \def\AdviceName{#2}%
2334   \advice@def@AdviceCname
2335   \let\AdviceReplaced\AdviceName
2336   \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.

```
2337 \advice@init@I{#1}{#2}%
```

All prepared, we execute the outer handler.

```
2338 \AdviceOuterHandler
2339 }
2340 \def\advice@def@AdviceCname{%
2341 \begingroup
2342 \escapechar=-1
2343 \expandafter\expandafter\expandafter\endgroup
2344 \expandafter\expandafter\expandafter\def
2345 \expandafter\expandafter\expandafter\AdviceCname
2346 \expandafter\expandafter\expandafter{\expandafter\string\AdviceName}%
2347 }
```

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

```
2348 \newif\ifAdviceRun
```

`\advice@default@outer@handler` The default outer handler merely executes the argument collector. Note that it works for both commands and environments.

```
2349 \def\advice@default@outer@handler{%
2350 \AdviceCollector
2351 }
```

`\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`.

```
2352 \def\advice@CollectArgumentsRaw{%
2353 \AdviceIfArgs{}{%
2354 \expandafter\GetDocumentCommandArgSpec\expandafter{\AdviceName}%
2355 \let\AdviceArgs\ArgumentSpecification
2356 }%
2357 \expanded{%
2358 \noexpand\CollectArgumentsRaw{%
2359 \noexpand\collargsCaller{\expandonce\AdviceName}%
2360 \expandonce\AdviceRawCollectorOptions
2361 \ifdefempty\AdviceCollectorOptions{}{%
2362 \noexpand\collargsSet{\expandonce\AdviceCollectorOptions}%
2363 }%
2364 }%
2365 {\expandonce\AdviceArgs}%
2366 {\expandonce\AdviceInnerHandler}%
2367 }%
2368 }
```

`\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”.)

```

2369 \def\advice@noargs@text{\advice@noargs}
2370 \def\AdviceIfArgs{%
2371   \ifx\AdviceArgs\advice@noargs@text
2372     \expandafter\@secondoftwo
2373   \else
2374     \expandafter\@firstoftwo
2375   \fi
2376 }

```

`\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`.

```

2377 \def\advice@pgfkeys@collector#1\pgfeov{%
2378   \AdviceInnerHandler{#1}%
2379 }

```

### 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.

```

2380 (*plain, context)
2381 \def\advice@activate@env#1#2{%
2382   \expanded{%
2383     \noexpand\advice@activate@cmd{#1}{\expandonce{\csname
2384 (context)      start%
2385               #2\endcsname}}%
2386   }%
2387 }
2388 \def\advice@deactivate@env#1#2{%
2389   \expanded{%
2390     \noexpand\advice@deactivate@cmd{#1}{\expandonce{\csname
2391 (context)      start%
2392               #2\endcsname}}%
2393   }%
2394 }
2395 (//plain, context)

```

We activate commands by redefining them; that’s the only way to do it. But we won’t activate a `LATEX` 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 `LATEX` environments by defining an outer handler for `\begin` (consequently, `LATEX` 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.

```

2396 (*latex)
2397 \def\advice@activate@env#1#2{%
2398   \ifcsdef{\advice@original@csname{#1}{#2}}{%
2399     \advice@activate@error@activated{#1}{#2}{Environment}{already}%
2400   }{%
2401     \csdef{\advice@original@csname{#1}{#2}}{%
2402       \PackageInfo{advice (#1)}{Activated environment "#2"}%
2403     }%

```

```

2404 }
2405 \def\advice@deactivate@env#1#2{%
2406   \ifcsdef{\advice@original@csname{#1}{#2}}{%
2407     \csundef{\advice@original@csname{#1}{#2}}{%
2408     }{%
2409       \advice@activate@error@activated{#1}{#2}{Environment}{not yet}%
2410       \PackageInfo{advice (#1)}{Deactivated environment "#2"}%
2411     }%
2412 }

```

`\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{\langle namespace \rangle}{\langle handled control sequence \rangle}`, so it grabs all these (#2 is the `\langle namespace \rangle` and #3 is the `\langle handled control sequence \rangle`, i.e. `\begin`) plus the environment name (#4).

```

2413 \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.

```

2414   \ifcsname\advice@original@csname{#2}{#4}\endcsname
2415     \expandafter\advice@begin@env@rc
2416   \else
2417     \expandafter\advice@original@cs
2418   \fi
2419   {#2}{#3}{#4}%
2420 }

```

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

```

2421 \def\advice@begin@env@rc#1#2#3{%

```

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

```

2422   \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).

```

2423   \AdviceRunConditions
2424   \ifAdviceRun
2425     \expandafter\advice@begin@env@outer
2426   \else
2427     \AdviceBailoutHandler
2428     \expandafter\advice@original@cs
2429   \fi
2430   {#1}{#2}{#3}%
2431 }

```

`\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.

```

2432 \def\advice@begin@env@outer#1#2#3{%
2433   \def\AdviceNamespace{#1}%
2434   \def\AdviceName{#3}%

```



```

2435 \let\AdviceCsname\advice@undefined
2436 \def\AdviceReplaced{#2{#3}}%
2437 \def\AdviceOriginal{\AdviceGetOriginal{#1}{#2}{#3}}%
2438 \advice@init@I{#1}{#3}%
2439 \AdviceOuterHandler
2440 }
2441 </latex>

```

### 8.1.5 Error messages

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

```

2442 \def\advice@activate@error@activated#1#2#3#4{%
2443 \ifadvice@activate@try
2444 \else
2445 \PackageError{advice (#1)}{#3 "\string#2" is #4 activated}{}%
2446 \fi
2447 }
2448 \def\advice@activate@error@undefined#1#2#3#4{%
2449 \ifadvice@activate@try
2450 \else
2451 \PackageError{advice (#1)}{%
2452 #3 "\string#2" you are trying to #4activate is not defined}{}%
2453 \fi
2454 }
2455 \def\advice@deactivate@error@changed#1#2{%
2456 \ifadvice@activate@try
2457 \else
2458 \PackageError{advice (#1)}{The definition of "\string#2" has changed since we
2459 have activated it. Has somebody overridden our command?}{If you have tried
2460 to deactivate so that you could immediately reactivate, you may want to try
2461 "force activate".}%
2462 \fi
2463 }
2464 \def\advice@error@advice@notcs#1#2{%
2465 \PackageError{advice}{The first argument of key "#1" should be either a single
2466 control sequence or an environment name, not "#2"}{}%
2467 }
2468 \def\advice@error@activate@notcsorenv#1#2{%
2469 \PackageError{advice}{Each item in the value of key "#1activate" should be
2470 either a control sequence or an environment name, not "#2".}{}%
2471 }
2472 \def\advice@error@storecs@notcs#1#2{%
2473 \PackageError{advice}{The value of key "#1" should be a single control sequence,
2474 not "\string#2"}{}%
2475 }
2476 \def\advice@error@noinnerhandler#1{%
2477 \PackageError{advice (\AdviceNamespace)}{The inner handler for
2478 "\expandafter\string\AdviceName" is not defined}{}%
2479 }

```

### 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`

```

2480 \def\AdviceTracingOn{%
2481 \let\advice@init@i\advice@trace@init@i
2482 \let\advice@init@I\advice@trace@init@I

```

```

2483 }
2484 \def\AdviceTracingOff{%
2485   \let\advice@init@i\advice@setup@init@i
2486   \let\advice@init@I\advice@setup@init@I
2487 }

```

`\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. Con $\TeX$ t, 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.

```

2488 (plain) \ifdefined\luatexversion
2489 (!latex) \long\def\advice@typeout#1{\directlua{texio.write_nl("\luaescapestring{#1}")}}
2490 (plain) \else
2491 (latex) \def\advice@typeout{\immediate\write\@unused}
2492 (plain) \def\advice@typeout{\immediate\write128}
2493 (plain) \fi
2494 \def\advice@trace#1{\advice@typeout{[tracing advice] #1}}

```

`\advice@trace@init@i` Install the tracing code.

```

\advice@trace@init@I
2495 \def\advice@trace@init@i#1#2{%
2496   \advice@trace{Advising \detokenize\expandafter{\string#2} (\detokenize{#1})}%
2497   \advice@trace{\space\space Original command meaning:
2498     \expandafter\expandafter\expandafter\meaning\advice@original@cs{#1}{#2}}%
2499   \advice@setup@init@i{#1}{#2}%
2500   \edef\AdviceRunConditions{%

```

We first execute the original run conditions, so that we can show the result.

```

2501   \expandonce\AdviceRunConditions
2502   \noexpand\advice@trace{\space\space
2503     Executing run conditions:
2504     \detokenize\expandafter{\AdviceRunConditions}
2505     -->
2506     \noexpand\ifAdviceRun true\noexpand\else false\noexpand\fi
2507   }%
2508 }%
2509 \edef\AdviceBailoutHandler{%
2510   \noexpand\advice@trace{\space\space
2511     Executing bailout handler:
2512     \detokenize\expandafter{\AdviceBailoutHandler}}%
2513   \expandonce\AdviceBailoutHandler
2514 }%
2515 }
2516 \def\advice@trace@init@I#1#2{%
2517   \advice@setup@init@I{#1}{#2}%
2518   \edef\AdviceOuterHandler{%
2519     \noexpand\advice@trace{\space\space
2520       Executing outer handler:
2521       \detokenize\expandafter{\AdviceOuterHandler}}%
2522     \expandonce\AdviceOuterHandler
2523   }%
2524   \edef\AdviceCollector{%
2525     \noexpand\advice@trace{\space\space
2526       Executing collector:
2527       \detokenize\expandafter{\AdviceCollector}}%
2528     \noexpand\advice@trace{\space\space\space\space
2529       Argument specification:
2530       \detokenize\expandafter{\AdviceArgs}}%
2531     \noexpand\advice@trace{\space\space\space\space
2532       Options:

```

```

2533     \detokenize\expandafter{\AdviceCollectorOptions}}%
2534     \noexpand\advice@trace{\space\space\space\space
2535     Raw options:
2536     \detokenize\expandafter{\AdviceRawCollectorOptions}}%
2537     \expandonce\AdviceCollector
2538 }%

```

The tracing inner handler must grab the provided argument, if it's to show what it is.

```

2539 \edef\advice@inner@handler@trace##1{%
2540     \noexpand\advice@trace{\space\space
2541     Executing inner handler:
2542     \detokenize\expandafter{\AdviceInnerHandler}}%
2543     \noexpand\advice@trace{\space\space\space\space
2544     Received arguments:
2545     \noexpand\detokenize{##1}}%
2546     \noexpand\advice@trace{\space\space\space\space
2547     Options:
2548     \detokenize\expandafter{\AdviceOptions}}%
2549     \expandonce{\AdviceInnerHandler}{##1}%
2550 }%
2551 \def\AdviceInnerHandler{\advice@inner@handler@trace}%
2552 }

```

```

2553 <plain>\resetatcatcode
2554 <context>\stopmodule
2555 <context>\protect
2556 </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.

```

2557 <*tikz>
2558 \edef\advice@resetatcatcode{\catcode`\noexpand\@the\catcode`\@relax}%
2559 \catcode`\@=11
2560 \def\AdviceCollectTikZArguments{%

```

We initialize the token register which will hold the collected arguments, and start the collection. Nothing of note happens until ...

```

2561     \toks0={}%
2562     \advice@tikz@anim
2563 }
2564 \def\advice@tikz@anim{%
2565     \pgfutil@ifnextchar[{\advice@tikz@opt}{%
2566         \pgfutil@ifnextchar:{\advice@tikz@anim@a}{%
2567             \advice@tikz@code}}%
2568 }
2569 \def\advice@tikz@anim@a#1=#2{%
2570     \toksapp0{#1={#2}}%
2571     \advice@tikz@anim
2572 }
2573 \def\advice@tikz@opt[#1]{%
2574     \toksapp0{[#1]}%
2575     \advice@tikz@code
2576 }

```

```

2577 \def\advice@tikz@code{%
2578   \pgfutil@ifnextchar\bgroup\advice@tikz@braced\advice@tikz@single
2579 }
2580 \long\def\advice@tikz@braced#1{\toksapp0{#{1}}\advice@tikz@done}
2581 \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.

```

2582 \def\advice@tikz@done{%
2583   \expandafter\AdviceInnerHandler\expandafter{\the\toks0}%
2584 }
2585 \adviceresetatcatcode
2586 </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  $\LaTeX$ , 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  $\TeX$  and  $\ConTeXt$  as well (same as Memoize), and, perhaps most importantly, to have the ability to collect the arguments verbatim.

### Identification

```

2587 <latex>\ProvidesPackage{collargs}[2024/01/02 v1.1.0 Collect arguments of any command]
2588 <context>%D \module[
2589 <context>%D   file=t-collargs.tex,
2590 <context>%D   version=1.1.0,
2591 <context>%D   title=CollArgs,
2592 <context>%D   subtitle=Collect arguments of any command,
2593 <context>%D   author=Saso Zivanovic,
2594 <context>%D   date=2024-01-02,
2595 <context>%D   copyright=Saso Zivanovic,
2596 <context>%D   license=LPPL,
2597 <context>%D ]
2598 <context>\writestatus{loading}{ConTeXt User Module / collargs}
2599 <context>\unprotect
2600 <context>\startmodule[collargs]

```

### Required packages

```

2601 <latex>\RequirePackage{pgfkeys}
2602 <plain>\input pgfkeys
2603 <context>\input t-pgfkey
2604 <latex>\RequirePackage{etoolbox}
2605 <plain,context>\input etoolbox-generic
2606 <plain>\edef\resetatcatcode{\catcode`\noexpand\@the\catcode`\@relax}
2607 <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
2608 \ifdefined\luatexversion
2609 \else
2610 \def\toksapp{\toks@cs@or@num\@toksapp}
2611 \def\gtoksapp{\toks@cs@or@num\@gtoksapp}
2612 \def\etoksapp{\toks@cs@or@num\@etoksapp}
2613 \def\xtoksapp{\toks@cs@or@num\@xtoksapp}
2614 \def\toks@cs@or@num#1#2#{%
```

Test whether `#2` (the original `#1`) is a number or a control sequence.

```
2615 \ifnum-2>-1#2
```

It is a number. `\toks@cs@or@num@num` will gobble `\toks@cs@or@num@cs` below.

```
2616 \expandafter\toks@cs@or@num@num
```

The register control sequence in `#2` is skipped over in the false branch.

```
2617 \fi
2618 \toks@cs@or@num@cs{#1}{#2}%
2619 }
```

`#1` is one of `\@toksapp` and friends. The second macro prefixes the register number by `\toks`.

```
2620 \def\toks@cs@or@num@cs#1#2{#1{#2}}
2621 \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.

```
2622 \long\def\@toksapp#1#2{#1\expandafter{\the#1#2}}%
2623 \long\def\@etoksapp#1#2{#1\expandafter{\expanded{\the#1#2}}}%
2624 \long\def\@gtoksapp#1#2{\global#1\expandafter{\the#1#2}}%
2625 \long\def\@xtoksapp#1#2{\global#1\expandafter{\expanded{\the#1#2}}}%
2626 \fi
```

`\CollectArguments` These are the only public commands provided by the package. `\CollectArguments` takes `\CollectArgumentsRaw` three arguments: the optional `#1` is the option list, processed 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.)

```
2627 \protected\def\CollectArguments{%
2628 \pgf@keys@utilifnextchar[\CollectArguments@i{\CollectArgumentsRaw{}}]%
2629 }
2630 \def\CollectArguments@i[#1]{\CollectArgumentsRaw{\collargsSet{#1}}}
2631 \protected\def\CollectArgumentsRaw#1#2#3{%
```

This group will be closed by `\collargs@`. once we grinded through the argument specification.

```
2632 \begingroup
```

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.

```
2633 \global\let\ifcollargs@last@verbatim\ifcollargs@verbatim
2634 \global\let\ifcollargs@last@verbatimbraces\ifcollargs@verbatimbraces
2635 \global\collargs@double@fixfalse
```

Apply the settings.

```
2636 \collargs@verbatim@wrap{#1}%
```

Initialize the space-grabber.

```
2637 \collargs@init@grabspaces
```

Remember the code to execute after collection.

```
2638 \def\collargs@next{#3}%
```

Initialize the token register holding the collected arguments.

```
2639 \global\collargs@toks{}%
```

Execute the central loop macro, which expects the argument specification #2 to be delimited from the following argument tokens by a dot.

```
2640 \collargs@#2.%
2641 }
```

`\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.

```
2642 \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.

```
2643 \def\collargs@cs@cases#1{\collargs@cs@cases@i#1\collargs@cs@cases@end}
2644 \let\collargs@cs@cases@end\relax
2645 \def\collargs@cs@cases@i{\futurelet\collargs@temp\collargs@cs@cases@ii}
2646 \def\collargs@cs@cases@ii#1#2\collargs@cs@cases@end{%
2647 \ifcat\noexpand\collargs@temp\relax
2648 \ifx\relax#2\relax
2649 \expandafter\expandafter\expandafter\@firstofthree
2650 \else
2651 \expandafter\expandafter\expandafter\@secondofthree
2652 \fi
2653 \else
2654 \expandafter\@thirdofthree
2655 \fi
2656 }
2657 \def\@firstofthree#1#2#3{#1}
2658 \def\@secondofthree#1#2#3{#2}
2659 \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.<sup>4</sup> 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 `caller=\foo`; the default is `caller=\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 `caller=\foo`, we effectively execute `\def\collargs@caller{\foo}`.

```

2660 \collargsSet{
2661   caller/.code={\collargsCaller{#1}},
2662 }
2663 \def\collargsCaller#1{%
2664   \collargs@cs@cases{#1}{%
2665     \let\collargs@temp\collargs@caller@cs
2666   }{%
2667     \let\collargs@temp\collargs@caller@csandmore
2668   }{%
2669     \let\collargs@temp\collargs@caller@env
2670   }%
2671   \collargs@temp{#1}%
2672 }
2673 \def\collargs@caller@cs#1{%

```

If `#1` is a single control sequence, just use that as the caller.

```

2674   \def\collargs@caller{#1}%
2675 }
2676 \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.

```

2677   \begingroup
2678   \escapechar -1
2679   \expandafter\endgroup
2680   \expandafter\def\expandafter\collargs@caller\expandafter{%
2681     \csname\string#1\endcsname
2682   }%
2683 }
2684 \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 `\begin{#1}` in  $\LaTeX$ .

```

2685   \expandafter\def\expandafter\collargs@caller\expandafter{%
2686     \csname
2687 <context>      start%
2688 <latex>       begin{%
2689              #1%
2690 <latex>       }%
2691              \endcsname
2692   }%
2693 }
2694 \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.

---

<sup>4</sup>The idea is borrowed from package `environ`, which is in turn based on code from `amsmath`.

```
2695 \newif\ifcollargs@verbatim
2696 \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` 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`.

```
2697 \collargsSet{
2698   verbatim/.code={\let\collargs@apply@verbatim\collargsVerbatim},
2699   verb/.code={\let\collargs@apply@verbatim\collargsVerb},
2700   no verbatim/.code={\let\collargs@apply@verbatim\collargsNoVerbatim},
2701 }
2702 \def\collargs@verbatim@wrap#1{%
2703   \let\collargs@apply@verbatim\relax
2704   #1%
2705   \collargs@apply@verbatim
2706 }
```

`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 `fix from no verbatim` 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 2707 \collargsSet{
2708   fix from verbatim/.code={\collargsFixFromVerbatim},
2709   fix from verb/.code={\collargsFixFromVerb},
2710   fix from no verbatim/.code={\collargsFixFromNoVerbatim},
2711 }
2712 \def\collargsFixFromNoVerbatim{%
2713   \global\collargs@fix@requestedtrue
2714   \global\let\ifcollargs@last@verbatim\iffalse
2715 }
2716 \def\collargsFixFromVerbatim{%
2717   \global\collargs@fix@requestedtrue
2718   \global\let\ifcollargs@last@verbatim\iftrue
2719   \global\let\ifcollargs@last@verbatimbraces\iftrue
2720 }
2721 \def\collargsFixFromVerb{%
2722   \global\collargs@fix@requestedtrue
2723   \global\let\ifcollargs@last@verbatim\iftrue
2724   \global\let\ifcollargs@last@verbatimbraces\iffalse
2725 }
```

`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.

```
2726 \collargsSet{
2727   braces/.code={\collargsBraces{#1}}%
2728 }
```

`environment` Set the environment name.

```
\collargsEnvironment
2729 \collargsSet{
```



```

2730 environment/.estore in=\collargs@b@envname
2731 }
2732 \def\collargsEnvironment#1{\edef\collargs@b@envname{#1}}
2733 \collargsEnvironment{}
```

**begin tag** When **begin tag**/**end tag** is in effect, the **begin**/**end-tag** will be will be prepended/ap-  
**end tag** pended to the environment body. **tags** is a shortcut for setting **begin tag** and **end tag**  
**tags** simultaneously.

```

\ifcollargsBeginTag
\ifcollargsEndTag
\ifcollargsAddTags
2734 \collargsSet{
2735   begin tag/.is if=collargsBeginTag,
2736   end tag/.is if=collargsEndTag,
2737   tags/.style={begin tag=#1, end tag=#1},
2738   tags/.default=true,
2739 }
2740 \newif\ifcollargsBeginTag
2741 \newif\ifcollargsEndTag
```

**ignore nesting** When this key is in effect, we will ignore any `\begin{<name>}`s and simply grab  
**ifcollargsIgnoreNesting** everything up to the first `\end{<name>}` (again, the markers are automatically adapted  
to the format).

```

2742 \collargsSet{
2743   ignore nesting/.is if=collargsIgnoreNesting,
2744 }
2745 \newif\ifcollargsIgnoreNesting
```

**ignore other tags** This key is only relevant in the non-verbatim and partial verbatim modes in L<sup>A</sup>T<sub>E</sub>X.  
**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`.

```

2746 \collargsSet{
2747   ignore other tags/.is if=collargsIgnoreOtherTags,
2748 }
2749 \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`.

```

2750 \collargsSet{
2751   append preprocessor/.code={\collargsAppendPreprocessor{#1}},
2752   prepend preprocessor/.code={\collargsPrependPreprocessor{#1}},
2753   append postprocessor/.code={\collargsAppendPostprocessor{#1}},
2754   prepend postprocessor/.code={\collargsPrependPostprocessor{#1}},
2755 }
2756 \def\collargsAppendPreprocessor{%
2757   \collargs@addprocessor\appto\collargs@preprocess@arg}
2758 \def\collargsPrependPreprocessor{%
2759   \collargs@addprocessor\preto\collargs@preprocess@arg}
2760 \def\collargsAppendPostprocessor{%
2761   \collargs@addprocessor\appto\collargs@postprocess@arg}
2762 \def\collargsPrependPostprocessor{%
2763   \collargs@addprocessor\preto\collargs@postprocess@arg}
```

Here, #1 will be either `\appto` or `\preto`, and #2 will be either `\collargs@preprocess@arg` or  
`\collargs@postprocess@arg`. #3 is the processor code.

```

2764 \def\collargs@addprocessor#1#2#3{%
2765   #1#2{%
2766     \expanded{%
2767       \unexpanded{#3}{\the\collargsArg}%
```

```

2768 }%
2769 }%
2770 }

```

`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.

```

2771 \def\collargs@preprocess@arg{}
2772 \def\collargs@postprocess@arg{}
2773 \collargsSet{
2774   clear preprocessors/.code={\collargsClearPreprocessors},
2775   clear postprocessors/.code={\collargsClearPostprocessors},
2776 }
2777 \def\collargsClearPreprocessors{\def\collargs@preprocess@arg{}}%
2778 \def\collargsClearPostprocessors{\def\collargs@postprocess@arg{}}%

```

`(append/prepend) expandable (pre/post)processor` These keys and macros simplify the definition of fully `\collargs(Append/Prepend)Expandable(Pre/Post)processor` expandable processors. Note that expandable processors are added to the same list as non-expandable processors.

```

2779 \collargsSet{
2780   append expandable preprocessor/.code={%
2781     \collargsAppendExpandablePreprocessor{#1}},
2782   prepend expandable preprocessor/.code={%
2783     \collargsPrependExpandablePreprocessor{#1}},
2784   append expandable postprocessor/.code={%
2785     \collargsAppendExpandablePostprocessor{#1}},
2786   prepend expandable postprocessor/.code={%
2787     \collargsPrependExpandablePostprocessor{#1}},
2788 }
2789 \def\collargsAppendExpandablePreprocessor{%
2790   \collargs@addeprocessor\appto\collargs@preprocess@arg}
2791 \def\collargsPrependExpandablePreprocessor{%
2792   \collargs@addeprocessor\preto\collargs@preprocess@arg}
2793 \def\collargsAppendExpandablePostprocessor{%
2794   \collargs@addeprocessor\appto\collargs@postprocess@arg}
2795 \def\collargsPrependExpandablePostprocessor{%
2796   \collargs@addeprocessor\preto\collargs@postprocess@arg}
2797 \def\collargs@addeprocessor#1#2#3{%
2798   #1#2{%
2799     \expanded{%
2800       \edef\noexpand\collargs@temp{\unexpanded{#3}{\the\collargsArg}}%
2801       \unexpanded{\expandafter\collargsArg\expandafter{\collargs@temp}}%
2802     }%
2803   }%
2804 }

```

`(append/prepend) (pre/post)wrap` These keys and macros simplify the definition of processors which yield `\collargs(Append/Prepend)(Pre/Post)wrap` the result after a single expansion. Again, they are added to the same list as other processors.

```

2805 \collargsSet{
2806   append prewrap/.code={\collargsAppendPrewrite{#1}},
2807   prepend prewrap/.code={\collargsPrependPrewrite{#1}},
2808   append postwrap/.code={\collargsAppendPostwrap{#1}},
2809   prepend postwrap/.code={\collargsPrependPostwrap{#1}},
2810 }
2811 \def\collargsAppendPrewrite{\collargs@addwrap\appto\collargs@preprocess@arg}
2812 \def\collargsPrependPrewrite{\collargs@addwrap\preto\collargs@preprocess@arg}
2813 \def\collargsAppendPostwrap{\collargs@addwrap\appto\collargs@postprocess@arg}
2814 \def\collargsPrependPostwrap{\collargs@addwrap\preto\collargs@postprocess@arg}
2815 \def\collargs@addwrap#1#2#3{%

```

```

2816 #1#2{%
2817   \long\def\collargs@temp##1{#3}%
2818   \expandafter\expandafter\expandafter\collargsArg
2819   \expandafter\expandafter\expandafter{%
2820     \expandafter\collargs@temp\expandafter{\the\collargsArg}%
2821   }%
2822 }%
2823 }

```

**no delimiters** When this conditional is in effect, the delimiter wrappers set by `\collargs@wrap` are `\ifcollargsNoDelimiters` ignored by `\collargs@appendarg`.

```

2824 \collargsSet{%
2825   no delimiters/.is if=collargsNoDelimiters,
2826 }
2827 \newif\ifcollargsNoDelimiters

```

**brace collected** When this conditional is set to false, the collected arguments are not enclosed in braces `\ifcollargsBraceCollected` when passed on to *<next-code>*.

```

2828 \collargsSet{%
2829   brace collected/.is if=collargsBraceCollected,
2830 }
2831 \newif\ifcollargsBraceCollected
2832 \collargsBraceCollectedtrue

```

## 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`.

```

2833 \def\collargs@{%
2834   \begingroup
2835   \collargs@@@
2836 }

```

`\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.

```

2837 \def\collargs@@@#1{%
2838   \collargs@in@{#1}{&+!>.%}
2839   \ifcollargs@in@
2840     \expandafter\collargs@@@iii
2841   \else
2842     \expandafter\collargs@@@i
2843   \fi
2844   #1%
2845 }
2846 \def\collargs@@@i#1.{%

```

Fix the category code of the next argument token, if necessary, and then proceed with the main loop.

```

2847   \collargs@fix{\collargs@@@i#1.%}
2848 }

```

Reset the fix request and set the last verbatim conditionals to the current state.

```

2849 \def\collargs@@@ii{%
2850   \global\collargs@fix@requestedfalse
2851   \global\let\ifcollargs@last@verbatim\ifcollargs@verbatim
2852   \global\let\ifcollargs@last@verbatimbraces\ifcollargs@verbatimbraces
2853   \collargs@@@iii
2854 }

```

Call the modifier or argument type handler denoted by the first token of the remainder of the argument specification.

```

2855 \def\collargs@@@iii#1{%
2856   \ifcsname collargs@#1\endcsname
2857     \csname collargs@#1\expandafter\endcsname
2858   \else

```

We throw an error if the token refers to no argument type or modifier.

```

2859     \collargs@error@badtype{#1}%
2860   \fi
2861 }

```

Throwing an error stops the processing of the argument specification, and closes the group opened in `\collargs@i`.

```

2862 \def\collargs@error@badtype#1#2.{%
2863   \PackageError{collargs}{Unknown xparse argument type or modifier "#1"
2864     for "\expandafter\string\collargs@caller\space"}{}%
2865   \endgroup
2866 }

```

`\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) `&`.

```

2867 \csdef{collargs@&}{%
2868   \futurelet\collargs@temp\collargs@amp@i
2869 }
2870 \def\collargs@amp@i{%

```

In ConT<sub>E</sub>Xt, `&` has character code “other” in the text.

```

2871 (!context) \ifx\collargs@temp&%
2872 (context)  \expandafter\ifx\detokenize{&}\collargs@temp
2873             \expandafter\collargs@amp@raw
2874   \else
2875     \expandafter\collargs@amp@set
2876   \fi
2877 }
2878 \def\collargs@amp@raw#1#2{%
2879   \collargs@verbatim@wrap{#2}%
2880   \collargs@@@
2881 }
2882 \def\collargs@amp@set#1{%
2883   \collargs@verbatim@wrap{\collargsSet{#1}}%
2884   \collargs@@@
2885 }

```

`\collargs@+` This modifier makes the next argument long, i.e. accept paragraph tokens.

```
2886 \csdef{collargs@+}{%
2887   \collargs@longtrue
2888   \collargs@@@
2889 }
2890 \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.)

```
2891 \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.

```
2892 \csdef{collargs@!}{%
2893   \collargs@grabspacesfalse
2894   \collargs@@@
2895 }
```

`\collargs@toks` 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.

```
2896 \newtoks\collargs@toks
```

`\collargsArg` An auxiliary, but publicly available token register, used for processing the argument, and by some argument type handlers.

```
2897 \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.

```
2898 \csdef{collargs@.}{%
```

Close the group opened in `\collargs@.`

```
2899   \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.

```
2900   \expanded{%
2901     \endgroup
2902     \noexpand\collargs@fix{%
2903       \expandonce\collargs@next
2904       \ifcollargsBraceCollected
2905         {\the\collargs@toks}%
2906       \else
2907         \the\collargs@toks
2908       \fi
2909     \collargs@spaces
2910   }%
2911 }%
2912 }
```

### 8.2.3 Auxiliary macros

`\collargs@appendarg` This macro is used by the argument type handlers to append the collected argument to the storage (`\collargs@toks`).

```
2913 \long\def\collargs@appendarg#1{%
```

Temporarily store the collected argument into a token register. The processors will manipulate the contents of this register.

```
2914 \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.

```
2915 \ifcollargs@double@fix
2916 \collargs@cancel@double@fix
2917 \fi
```

Process the argument with user-definable preprocessors, the wrapper defined by the argument type, and user-definable postprocessors.

```
2918 \collargs@preprocess@arg
2919 \ifcollargsNoDelimiters
2920 \else
2921 \collargs@process@arg
2922 \fi
2923 \collargs@postprocess@arg
```

Append the processed argument, preceded by any grabbed spaces (in the correct mode), to the storage.

```
2924 \xtoksapp\collargs@toks{\collargs@grabbed@spaces\the\collargsArg}%
```

Initialize the space-grabber.

```
2925 \collargs@init@grabspaces
```

Once the argument was appended to the list, we can close its group, opened by `\collargs@`.

```
2926 \endgroup
2927 }
```

`\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.

```
2928 \def\collargs@wrap{\collargs@addwrap\appto\collargs@process@arg}
2929 \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.

```
2930 \def\collargs@defcollector#1#1{%
2931 \ifcollargs@long\long\fi
```

```

2932 \expandafter\def\collargs@caller#1%
2933 }
2934 \def\collargs@defusecollector#1#{%
2935 \afterassignment\collargs@caller
2936 \ifcollargs@long\long\fi
2937 \expandafter\def\collargs@caller#1%
2938 }
2939 \def\collargs@letusecollector#1{%
2940 \expandafter\let\collargs@caller#1%
2941 \collargs@caller
2942 }
2943 \newif\ifcollargs@grabspaces
2944 \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.

```

2945 \def\collargs@init@grabspaces{%
2946 \gdef\collargs@gs@state{0}%
2947 \gdef\collargs@spaces{}%
2948 \gdef\collargs@otherspaces{}%
2949 }

```

`\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.

```

2950 \def\collargs@grabspaces#1{%
2951 \edef\collargs@gs@next{\unexpanded{#1}}%
2952 \ifnum\collargs@gs@state=0
2953 \gdef\collargs@gs@state{1}%
2954 \expandafter\collargs@gs@i
2955 \else
2956 \expandafter\collargs@gs@next
2957 \fi
2958 }
2959 \def\collargs@gs@i{%
2960 \futurelet\collargs@temp\collargs@gs@g
2961 }

```

We check for grouping characters even in the verbatim mode, because we might be in the partial verbatim.

```

2962 \def\collargs@gs@g{%
2963 \ifcat\noexpand\collargs@temp\bgroup
2964 \expandafter\collargs@gs@next
2965 \else
2966 \ifcat\noexpand\collargs@temp\egroup
2967 \expandafter\expandafter\expandafter\collargs@gs@next
2968 \else
2969 \expandafter\expandafter\expandafter\collargs@gs@ii
2970 \fi
2971 \fi
2972 }
2973 \def\collargs@gs@ii{%
2974 \ifcollargs@verbatim
2975 \expandafter\collargs@gs@iii
2976 \else

```

```

2977   \expandafter\collargs@gs@iii
2978   \fi
2979 }

```

This works because the character code of a space token is always 32.

```

2980 \def\collargs@gs@iii{%
2981   \expandafter\ifx\space\collargs@temp
2982   \expandafter\collargs@gs@iv
2983   \else
2984   \expandafter\collargs@gs@next
2985   \fi
2986 }
2987 \expandafter\def\expandafter\collargs@gs@iv\space{%
2988   \gappto\collargs@spaces{ }%
2989   \xappto\collargs@otherspaces{\collargs@otherspace}%
2990   \collargs@gs@i
2991 }

```

We need the space of category 12 above.

```

2992 \begingroup\catcode\ =12\relax\gdef\collargs@otherspace{ }\endgroup
2993 \def\collargs@gs@iii#1{%

```

Macro `\collargs@cc` recalls the “outside” category code of character `#1`; see section [8.2.5](#).

```

2994   \ifnum\collargs@cc{#1}=10

```

We have a space.

```

2995   \expandafter\collargs@gs@iv
2996   \else
2997   \ifnum\collargs@cc{#1}=5

```

We have a newline.

```

2998   \expandafter\expandafter\expandafter\collargs@gs@v
2999   \else
3000   \expandafter\expandafter\expandafter\collargs@gs@next
3001   \fi
3002   \fi
3003   #1%
3004 }
3005 \def\collargs@gs@iv#1{%
3006   \gappto\collargs@otherspaces{#1}%

```

No matter how many verbatim spaces we collect, they equal a single non-verbatim space.

```

3007   \gdef\collargs@spaces{ }%
3008   \collargs@gs@i
3009 }
3010 \def\collargs@gs@v{%

```

Only add the first newline.

```

3011   \ifnum\collargs@gs@state=2
3012   \expandafter\collargs@gs@next
3013   \else
3014   \expandafter\collargs@gs@vi
3015   \fi
3016 }
3017 \def\collargs@gs@vi#1{%
3018   \gdef\collargs@gs@state{2}%
3019   \gappto\collargs@otherspaces{#1}%
3020   \gdef\collargs@spaces{ }%
3021   \collargs@gs@i
3022 }

```



`\collargs@maybegrabspace` This macro grabs any following spaces, but it will do so only when conditional `\ifcollargs@grabspace`, which can be *unset* by modifier `!`, is in effect. The macro is used by handlers for types `d` and `t`.

```

3023 \def\collargs@maybegrabspace{%
3024   \ifcollargs@grabspace
3025     \expandafter\collargs@grabspace
3026   \else
3027     \expandafter\@firstofone
3028   \fi
3029 }

```

`\collargs@grabbed@space` 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.

```

3030 \def\collargs@grabbed@space{%
3031   \ifcollargs@verbatim
3032     \collargs@otherspace
3033   \else
3034     \collargs@space
3035   \fi
3036 }

```

`\collargs@reinsert@space` 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.

```

3037 \def\collargs@reinsert@space#1{%
3038   \expanded{%
3039     \unexpanded{%
3040       \collargs@init@grabspace
3041       #1%
3042     }%
3043     \collargs@grabbed@space
3044   }%
3045 }

```

`\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`.

```

3046 \long\def\collargs@ifnextcat#1#2#3{%
3047   \let\pgf@keys@utilreserved@d=#1%
3048   \def\pgf@keys@utilreserved@a{#2}%
3049   \def\pgf@keys@utilreserved@b{#3}%
3050   \futurelet\pgf@keys@utillet@token\collargs@ifncat}
3051 \def\collargs@ifncat{%
3052   \ifx\pgf@keys@utillet@token\pgf@keys@utilstoken
3053     \let\pgf@keys@utilreserved@c\collargsxifnch
3054   \else
3055     \ifcat\noexpand\pgf@keys@utillet@token\pgf@keys@utilreserved@d
3056       \let\pgf@keys@utilreserved@c\pgf@keys@utilreserved@a
3057     \else
3058       \let\pgf@keys@utilreserved@c\pgf@keys@utilreserved@b
3059     \fi
3060   \fi
3061   \pgf@keys@utilreserved@c}
3062 {%
3063   \def\:\{\collargs@xifncat}
3064   \expandafter\gdef\:\{\futurelet\pgf@keys@utillet@token\collargs@ifncat}
3065 }

```

`\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.

```

3066 \def\collargs@forrange#1#2{%
3067   \expanded{%
3068     \noexpand\collargs@forrange@i{\number#1}{\number#2}%
3069   }%
3070 }
3071 \def\collargs@forrange@i#1#2{%
3072   \ifnum#1>#2 %
3073     \expandafter\@gobble
3074   \else
3075     \expandafter\@firstofone
3076   \fi
3077   {%
3078     \collargs@do{#1}%
3079     \expandafter\collargs@forrange@i\expandafter{\number\numexpr#1+1\relax}{#2}%
3080   }%
3081 }

```

`\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.

```

3082 \def\collargs@forranges{\forcsvlist\collarg@forrange@i}
3083 \def\collarg@forrange@i#1{\collarg@forrange@ii#1-}
3084 \def\collarg@forrange@ii#1-#2-{\collargs@forrange{#1}{#2}}

```

`\collargs@percentchar` This macro holds the percent character of category 12.

```

3085 \begingroup
3086 \catcode`\%=12
3087 \gdef\collargs@percentchar{%
3088 \endgroup

```

## 8.2.4 The handlers

`\collargs@l` We will first define the handler for the very funky argument type 1, which corresponds to T<sub>E</sub>X'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.

```
3089 \def\collargs@l#1. {%
```

Any pre-grabbed spaces in fact belong into the argument.

```

3090   \collargs@reinsert@spaces{\collargs@l@i#1.}%
3091 }
3092 \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.

```
3093   \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.

```

3094   \ifcollargs@verbatimbraces
3095     \expandafter\collargs@l@verb
3096   \else
3097     \expandafter\collargs@l@ii
3098   \fi
3099 }

```

We grab the rest of the argument specification (`#1`), to be reinserted into the token stream when we reexecute the central loop.

```
3100 \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.

```
3101 \collargs@defusecollector##1##{%
```

We append the collected argument, `##1`, to `\collargs@toks`, the token register holding the collected argument tokens.

```
3102 \collargs@appendarg{##1}%
```

Back to the central loop, with the rest of the argument specification reinserted.

```
3103 \collargs@#1.%
```

```
3104 }%
```

```
3105 }
```

```
3106 \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.

```
3107 \expandafter\collargs@defusecollector
```

```
3108 \expandafter##\expandafter1\collargs@other@bgroup{%
```

Appending the argument works the same as in the non-verbatim case.

```
3109 \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.

```
3110 \expanded{%
```

```
3111 \noexpand\collargs@\unexpanded{#1.}%
```

```
3112 \collargs@other@bgroup
```

```
3113 }%
```

```
3114 }%
```

```
3115 }
```

`\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.

```
3116 \def\collargs@u{%
```

```
3117 \ifcollargs@verbatim
```

```
3118 \expandafter\collargs@u@verb
```

```
3119 \else
```

```
3120 \expandafter\collargs@u@i
```

```
3121 \fi
```

```
3122 }
```

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`.

```
3123 \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.

```
3124 \expandafter\collargs@u@i\expandafter{\detokenize\expandafter{\string#1}}%
3125 }
```

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.

```
3126 \def\collargs@u@i#1#2.{%
3127 \collargs@reinsert@spaces{\collargs@u@ii{#1}#2.}%
3128 }
3129 \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 `l`, 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`).

```
3130 \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).

```
3131 \collargs@wrap{###1#1}%
```

Expand the first token in `##1`, which we know to be `\collargs@empty`, with empty expansion.

```
3132 \expandafter\collargs@appendarg\expandafter{##1}%
3133 \collargs@#2.%
3134 }%
```

Insert `\collargs@empty` into the input stream, in front of the “real” argument tokens.

```
3135 \collargs@empty
3136 }
3137 \def\collargs@empty{}
```

`\collargs@r` Finally, a real argument type: required delimited argument.

```
3138 \def\collargs@r{%
3139 \ifcollargs@verbatim
3140 \expandafter\collargs@r@verb
3141 \else
3142 \expandafter\collargs@r@i
3143 \fi
3144 }
3145 \def\collargs@r@verb#1#2{%
3146 \expandafter\collargs@r@i\detokenize{#1#2}%
3147 }
3148 \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.

```
3149 \collargs@defcollector#1{%
```

The inner collector will grab everything up to the closing delimiter.

```
3150 \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`.

```
3151 \collargs@wrap{#1#####1#2}%
3152 \expandafter\collargs@appendarg\expandafter{####1}%
3153 \collargs@#3.%
3154 }%
3155 \collargs@empty
3156 }%
```

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.

```
3157 \collargs@grabspaces\collargs@caller
3158 }
```

`\collargs@R` Discard the default and execute `r`.

```
3159 \def\collargs@R#1#2#3{\collargs@r#1#2}
```

`\collargs@d` Optional delimited argument. Very similar to `r`.

```
3160 \def\collargs@d{%
3161 \ifcollargs@verbatim
3162 \expandafter\collargs@d@verb
3163 \else
3164 \expandafter\collargs@d@i
3165 \fi
3166 }
3167 \def\collargs@d@verb#1#2{%
3168 \expandafter\collargs@d@i\detokenize{#1#2}%
3169 }
3170 \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.

```
3171 \def\collargs@d@noopt{%
3172 \global\collargs@fix@requestedtrue
3173 \endgroup
3174 \collargs@#3.%
3175 }%
```

The collector(s) are exactly as for `r`.

```
3176 \collargs@defcollector#1{%
3177 \collargs@defusecollector####1#2{%
3178 \collargs@wrap{#1#####1#2}%
3179 \expandafter\collargs@appendarg\expandafter{####1}%
3180 \collargs@#3.%
3181 }%
3182 \collargs@empty
3183 }%
```

This macro will check, in conjunction with `\futurelet` below, whether the optional argument is present or not.

```

3184 \def\collargs@d@ii{%
3185   \ifx#1\collargs@temp
3186   \expandafter\collargs@caller
3187   \else
3188   \expandafter\collargs@d@noopt
3189   \fi
3190 }%
```

Whether spaces are allowed in front of this type of argument depends on the presence of modifier `!`.

```

3191 \collargs@maybegrabspaces{\futurelet\collargs@temp\collargs@d@ii}%
3192 }
```

`\collargs@D` Discard the default and execute `d`.

```

3193 \def\collargs@D#1#2#3{\collargs@d#1#2}
```

`\collargs@o` `o` is just `d` with delimiters `[` and `]`.

```

3194 \def\collargs@o{\collargs@d[]}
```

`\collargs@O` `O` is just `d` with delimiters `[` and `]` and the discarded default.

```

3195 \def\collargs@O#1{\collargs@d[]}
```

`\collargs@t` An optional token. Similar to `d`.

```

3196 \def\collargs@t{%
3197   \ifcollargs@verbatim
3198   \expandafter\collargs@t@verb
3199   \else
3200   \expandafter\collargs@t@i
3201   \fi
3202 }
3203 \def\collargs@t@space{ }
3204 \def\collargs@t@verb#1{%
3205   \let\collargs@t@space\collargs@otherspace
3206   \expandafter\collargs@t@i\expandafter{\detokenize{#1}}%
3207 }
3208 \def\collargs@t@i#1{%
3209   \expandafter\ifx\space#1%
3210   \expandafter\collargs@t@s
3211   \else
3212   \expandafter\collargs@t@I\expandafter#1%
3213   \fi
3214 }
3215 \def\collargs@t@s#1.{%
3216   \collargs@grabspaces{%
3217     \ifcollargs@grabspaces
3218     \collargs@appendarg{}}%
3219   \else
3220   \expanded{%
3221     \noexpand\collargs@init@grabspaces
3222     \noexpand\collargs@appendarg{\collargs@grabbed@spaces}}%
3223   }%
3224   \fi
3225   \collargs@#1.%
3226 }%
3227 }
```

```

3228 \def\collargs@t@I#1#2.{%
3229   \def\collargs@t@noopt{%
3230     \global\collargs@fix@requestedtrue
3231     \endgroup
3232     \collargs@#2.%
3233   }%
3234 \def\collargs@t@opt##1{%
3235   \collargs@appendarg{#1}%
3236   \collargs@#2.%
3237   }%
3238 \def\collargs@t@ii{%
3239   \ifx#1\collargs@temp
3240     \expandafter\collargs@t@opt
3241   \else
3242     \expandafter\collargs@t@noopt
3243   \fi
3244   }%
3245 \collargs@maybegrabspaces{\futurelet\collargs@temp\collargs@t@ii}%
3246 }
3247 \def\collargs@t@opt@space{%
3248   \expanded{\noexpand\collargs@t@opt{\space}\expandafter}\romannumeral-0%
3249   }%

```

`\collargs@s` The optional star is just a special case of `t`.

```

3250 \def\collargs@s{\collargs@t*}

```

`\collargs@m` Mandatory argument. Interestingly, here's where things get complicated, because we have to take care of several  $\TeX$  quirks.

```

3251 \def\collargs@m{%
3252   \ifcollargs@verbatim
3253     \expandafter\collargs@m@verb
3254   \else
3255     \expandafter\collargs@m@i
3256   \fi
3257 }

```

The non-verbatim mode. First, collect any spaces in front of the argument.

```

3258 \def\collargs@m@i#1.{%
3259   \collargs@grabspaces{\collargs@m@checkforgroup#1.}%
3260 }

```

Is the argument in braces or not?

```

3261 \def\collargs@m@checkforgroup#1.{%
3262   \edef\collargs@action{\unexpanded{\collargs@m@checkforgroup@i#1.}}%
3263   \futurelet\collargs@token\collargs@action
3264 }
3265 \def\collargs@m@checkforgroup@i{%
3266   \ifcat\noexpand\collargs@token\bgroup
3267     \expandafter\collargs@m@group
3268   \else
3269     \expandafter\collargs@m@token
3270   \fi
3271 }

```

The argument is given in braces, so we put them back around it (`\collargs@wrap`) when appending to the storage.

```

3272 \def\collargs@m@group#1.{%
3273   \collargs@defusecollector##1{%

```

```

3274 \collargs@wrap{####1}%
3275 \collargs@appendarg{##1}%
3276 \collargs@#1.%
3277 }%
3278 }

```

The argument is a single token, we append it to the storage as is.

```

3279 \def\collargs@m@token#1.{%
3280 \collargs@defusecollector##1{%
3281 \collargs@appendarg{##1}%
3282 \collargs@#1.%
3283 }%
3284 }

```

The verbatim mode. Again, we first collect any spaces in front of the argument.

```

3285 \def\collargs@m@verb#1.{%
3286 \collargs@grabspace{\collargs@m@verb@checkforgroup#1.}%
3287 }

```

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.

```

3288 \def\collargs@m@verb@checkforgroup{%
3289 \ifcollargs@verbatimbraces
3290 \expandafter\collargs@m@verb@checkforgroup@i
3291 \else
3292 \expandafter\collargs@m@checkforgroup
3293 \fi
3294 }

```

Is the argument in verbatim braces?

```

3295 \def\collargs@m@verb@checkforgroup@i#1.{%
3296 \def\collargs@m@verb@checkforgroup@ii{\collargs@m@verb@checkforgroup@iii#1.}%
3297 \futurelet\collargs@temp\collargs@m@verb@checkforgroup@ii
3298 }
3299 \def\collargs@m@verb@checkforgroup@iii#1.{%
3300 \expandafter\ifx\collargs@other@bgroup\collargs@temp

```

Yes, the argument is in (verbatim) braces.

```

3301 \expandafter\collargs@m@verb@group
3302 \else

```

We need to manually check whether the following token is a (verbatim) closing brace, and throw an error if it is.

```

3303 \expandafter\ifx\collargs@other@egroup\collargs@temp
3304 \expandafter\expandafter\expandafter\collargs@m@verb@egrouperror
3305 \else

```

The argument is a single token.

```

3306 \expandafter\expandafter\expandafter\collargs@m@v@token
3307 \fi
3308 \fi
3309 #1.%
3310 }
3311 \def\collargs@m@verb@egrouperror#1.{%
3312 \PackageError{collargs}{%
3313 Argument of \expandafter\string\collargs@caller\space has an extra
3314 \iffalse{\else\string}}{}}%
3315 }

```



A single-token verbatim argument.

```
3316 \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.)

```
3317 \ifnum\collargs@cc{#2}=0
3318   \expandafter\collargs@m@v@token@cs
3319 \else
3320   \expandafter\collargs@m@token
3321 \fi
3322 #1.#2%
3323 }
```

Is it a one-character control sequence?

```
3324 \def\collargs@m@v@token@cs#1.#2#3{%
3325   \ifnum\collargs@cc{#3}=11
3326     \expandafter\collargs@m@v@token@cs@letter
3327   \else
3328     \expandafter\collargs@m@v@token@cs@nonletter
3329   \fi
3330 #1.#2#3%
3331 }
```

Store `\<token>`.

```
3332 \def\collargs@m@v@token@cs@nonletter#1.#2#3{%
3333   \collargs@appendarg{#2#3}%
3334   \collargs@#1.%
3335 }
```

Store `\` to a temporary register, we’ll parse the control sequence name now.

```
3336 \def\collargs@m@v@token@cs@letter#1.#2{%
3337   \collargsArg{#2}%
3338   \def\collargs@tempa{#1}%
3339   \collargs@m@v@token@cs@letter@i
3340 }
```

Append a letter to the control sequence.

```
3341 \def\collargs@m@v@token@cs@letter@i#1{%
3342   \ifnum\collargs@cc{#1}=11
3343     \toksapp\collargsArg{#1}%
3344     \expandafter\collargs@m@v@token@cs@letter@i
3345   \else
```

Finish, returning the non-letter to the input stream.

```
3346     \expandafter\collargs@m@v@token@cs@letter@ii\expandafter#1%
3347   \fi
3348 }
```

Store the verbatim control sequence.

```
3349 \def\collargs@m@v@token@cs@letter@ii{%
3350   \expanded{%
3351     \unexpanded{%
3352       \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
3353     }%
3354     \noexpand\collargs@\expandonce\collargs@tempa.%
3355   }%
3356 }
```

The verbatim mandatory argument is delimited by verbatim braces. We have to use the heavy machinery adapted from `cprotect`.

```

3357 \def\collargs@m@verb@group#1.#2{%
3358   \let\collargs@begintag\collargs@other@bgroup
3359   \let\collargs@endtag\collargs@other@egroup
3360   \def\collargs@tagarg{}%
3361   \def\collargs@commandatend{\collargs@m@verb@group@i#1.}%
3362   \collargs@readContent
3363 }

```

This macro appends the result given by the heavy machinery, waiting for us in macro `\collargsArg`, to `\collargs@toks`, but not before dressing it up (via `\collargs@wrap`) in a pair of verbatim braces.

```

3364 \def\collargs@m@verb@group@i{%
3365   \edef\collargs@temp{%
3366     \collargs@other@bgroup\unexpanded{##1}\collargs@other@egroup}%
3367   \expandafter\collargs@wrap\expandafter{\collargs@temp}%
3368   \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
3369   \collargs@
3370 }

```

`\collargs@g` An optional group: same as `m`, but we simply bail out if we don't find the group character.

```

3371 \def\collargs@g{%
3372   \def\collargs@m@token{%
3373     \global\collargs@fix@requestedtrue
3374     \endgroup
3375     \collargs@
3376   }%
3377   \let\collargs@m@v@token\collargs@m@token
3378   \collargs@m
3379 }

```

`\collargs@G` Discard the default and execute `g`.

```

3380 \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.

```

3381 \def\collargs@v#1.{%
3382   \begingroup
3383   \collargsBraces{}}%
3384   \collargsVerbatim
3385   \collargs@grabspaces{\collargs@v@i#1.}%
3386 }
3387 \def\collargs@v@i#1.#2{%
3388   \expandafter\ifx\collargs@other@bgroup#2%

```

If the first token we see is an opening brace, use the `cprotect` adaptation to grab the group.

```

3389   \let\collargs@begintag\collargs@other@bgroup
3390   \let\collargs@endtag\collargs@other@egroup
3391   \def\collargs@tagarg{}%
3392   \def\collargs@commandatend{%
3393     \edef\collargs@temp{%
3394       \collargs@other@bgroup\unexpanded{####1}\collargs@other@egroup}%
3395     \expandafter\collargs@wrap\expandafter{\collargs@temp}%
3396     \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%
3397     \endgroup
3398     \collargs@#1.%

```

```

3399 }%
3400 \expandafter\collargs@readContent
3401 \else

```

Otherwise, the verbatim argument is delimited by two identical characters (#2).

```

3402 \collargs@defcollector##1#2{%
3403 \collargs@wrap{#2###1#2}%
3404 \collargs@appendarg{##1}%
3405 \endgroup
3406 \collargs@#1.%
3407 }%
3408 \expandafter\collargs@caller
3409 \fi
3410 }

```

`\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).

```

3411 \def\collargs@b{%
3412 \collargs@ifnextcat\bgroup\collargs@bg\collargs@bi
3413 }
3414 \def\collargs@bg#1{%
3415 \edef\collargs@b@envname{#1}%
3416 \collargs@bi
3417 }
3418 \def\collargs@bi#1.{%

```

Convert the environment name to verbatim if necessary.

```

3419 \ifcollargs@verbatim
3420 \edef\collargs@b@envname{\detokenize\expandafter{\collargs@b@envname}}%
3421 \fi

```

This is a format-specific macro which sets up `\collargs@begintag` and `\collargs@endtag`.

```

3422 \collargs@bi@defCPTbeginend
3423 \edef\collargs@tagarg{%
3424 \ifcollargs@verbatimbraces
3425 \else
3426 \ifcollargsIgnoreOtherTags
3427 \collargs@b@envname
3428 \fi
3429 \fi
3430 }%

```

Run this after collecting the body.

```

3431 \def\collargs@commandatend{%

```

In  $\text{\LaTeX}$ , we might, depending on the verbatim mode, need to check whether the environment name is correct.

```

3432 <latex> \collargs@bii

```

In plain  $\text{\TeX}$  and  $\text{\ConTeXt}$ , we can skip directly to `\collargs@biii`.

```

3433 <plain, context> \collargs@biii
3434 #1.%
3435 }%

```

Collect the environment body, but first, put any grabbed spaces back into the input stream.

```
3436 \collargs@reinsert@spaces\collargs@readContent
3437 }
3438 <*/latex>
```

In L<sup>A</sup>T<sub>E</sub>X 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>}`.

```
3439 \def\collargs@bi@defCPTbeginend{%
3440 \edef\collargs@begintag{%
3441 \ifcollargs@verbatim
3442 \expandafter\string
3443 \else
3444 \expandafter\noexpand
3445 \fi
3446 \begin
3447 \ifcollargs@verbatimbraces
3448 \collargs@other@bgroup\collargs@b@envname\collargs@other@egroup
3449 \fi
3450 }%
3451 \edef\collargs@endtag{%
3452 \ifcollargs@verbatim
3453 \expandafter\string
3454 \else
3455 \expandafter\noexpand
3456 \fi
3457 \end
3458 \ifcollargs@verbatimbraces
3459 \collargs@other@bgroup\collargs@b@envname\collargs@other@egroup
3460 \fi
3461 }%
3462 }
3463 </latex>
3464 <*plain, context>
```

We can search for the entire `\<name>/\end<name>` (in T<sub>E</sub>X) or `\start<name>/\stop<name>` (in ConT<sub>E</sub>Xt), either as a control sequence (in the regular mode), or as a string (in the verbatim modes).

```
3465 \def\collargs@bi@defCPTbeginend{%
3466 \edef\collargs@begintag{%
3467 \ifcollargs@verbatim
3468 \expandafter\expandafter\expandafter\string
3469 \else
3470 \expandafter\expandafter\expandafter\noexpand
3471 \fi
3472 \csname
3473 <context> start%
3474 \collargs@b@envname
3475 \endcsname
3476 }%
3477 \edef\collargs@endtag{%
3478 \ifcollargs@verbatim
3479 \expandafter\expandafter\expandafter\string
3480 \else
3481 \expandafter\expandafter\expandafter\noexpand
3482 \fi
3483 \csname
3484 <plain> end%
```

```

3485 <context>      stop%
3486              \collargs@b@envname
3487              \endcsname
3488          }%
3489      }
3490 </plain, context>
3491 <*latex>

```

Check whether we're in front of the (braced) environment name (in L<sup>A</sup>T<sub>E</sub>X), and consume it.

```

3492 \def\collargs@bii{%
3493   \ifcollargs@verbatimbraces
3494     \expandafter\collargs@biii
3495   \else
3496     \ifcollargsIgnoreOtherTags

```

We shouldn't check the name in this case, because it was already checked, and consumed.

```

3497     \expandafter\expandafter\expandafter\collargs@biii
3498   \else
3499     \expandafter\expandafter\expandafter\collargs@b@checkend
3500   \fi
3501 \fi
3502 }
3503 \def\collargs@b@checkend#1.{%
3504   \collargs@grabspaces{\collargs@b@checkend@i#1.}%
3505 }
3506 \def\collargs@b@checkend@i#1.#2{%
3507   \def\collargs@temp{#2}%
3508   \ifx\collargs@temp\collargs@b@envname
3509   \else
3510     \collargs@b@checkend@error
3511   \fi
3512   \collargs@biii#1.%
3513 }
3514 \def\collargs@b@checkend@error{%
3515   \PackageError{collargs}{Environment "\collargs@b@envname" ended as
3516     "\collargs@temp"}{}%
3517 }
3518 </latex>

```

This macro stores the collected body.

```

3519 \def\collargs@biii{%

```

Define the wrapper macro (`\collargs@temp`).

```

3520   \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.

```

3521   \expandafter\collargs@appendarg\expandafter{\the\collargsArg}%

```

Reexecute the central loop.

```

3522   \collargs@
3523 }
3524 \def\collargs@b@def@wrapper{%
3525 <latex>   \edef\collargs@temp{\collargs@b@envname}}%
3526   \edef\collargs@temp{%

```

Was the begin-tag requested?

```

3527   \ifcollargsBeginTag

```

`\collargs@begintag` is already adapted to the format and the verbatim mode.

```
3528 \expandonce\collargs@begintag
```

Add the braced environment name in  $\LaTeX$  in the regular and partial verbatim mode.

```
3529 {*latex}
3530 \ifcollargs@verbatimbraces\else\collargs@temp\fi
3531 {/latex}
3532 \fi
```

This is the body.

```
3533 #####1%
```

Rinse and repeat for the end-tag.

```
3534 \ifcollargsEndTag
3535 \expandonce\collargs@endtag
3536 {*latex}
3537 \ifcollargs@verbatimbraces\else\collargs@temp\fi
3538 {/latex}
3539 \fi
3540 }%
3541 \expandafter\collargs@wrap\expandafter{\collargs@temp}%
3542 }
```

`\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.

```
3543 \def\collargs@readContent{%
```

Define macro which will search for the first begin-tag.

```
3544 \ifcollargs@long\long\fi
3545 \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.

```
3546 \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.

```
3547 \futurelet\collargs@temp\collargs@gobbleOneB@i
3548 }%
```

Define macro which will search for the first end-tag. We make it long if so required (by +).

```
3549 \ifcollargs@long\long\fi
3550 \collargs@CPT@def\collargs@gobbleUntilE\collargs@endtag{%
```

Expand `\collargs@empty` at the start of `##1`.

```
3551 \expandafter\toksapp\expandafter0\expandafter{##1}%
3552 \collargs@gobbleUntilE@i
3553 }%
```

Initialize.

```
3554 \collargs@begins=0\relax
3555 \collargsArg{}%
3556 \toks0{}%
```

We will call `\collargs@gobbleUntilE` via the caller control sequence.

```
3557 \collargs@letusecollector\collargs@gobbleUntilE
```

We insert `\collargs@empty` to avoid the potential debracing problem.

```
3558 \collargs@empty
3559 }
```

How many begin-tags do we have opened?

```
3560 \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.

```
3561 \def\collargs@CPT@def#1#2{%
3562 \expandafter\def\expandafter#1%
3563 \expandafter##\expandafter1#2%
3564 }
```

A quark quard.

```
3565 \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.

```
3566 \def\collargs@gobbleOneB@i{%
3567 \def\collargs@begins@increment{1}%
3568 \ifx\collargs@qend\collargs@temp
```

We have reached the fake begin-tag. Note that we found the end-tag.

```
3569 \def\collargs@begins@increment{-1}%
```

Gobble the quark guard.

```
3570 \expandafter\collargs@gobbleOneB@v
3571 \else
```

Append the real begin-tag to the temporary tokens.

```
3572 \etoksapp0{\expandonce\collargs@begintag}%
3573 \expandafter\collargs@gobbleOneB@ii
3574 \fi
3575 }%
```

Do we have to check the tag argument (i.e. the environment name after `\begin`)?

```
3576 \def\collargs@gobbleOneB@ii{%
3577 \expandafter\ifx\expandafter\relax\collargs@tagarg\relax
3578 \expandafter\collargs@gobbleOneB@vi
3579 \else
```

Yup, so let's (carefully) collect the tag argument.

```
3580 \expandafter\collargs@gobbleOneB@iii
3581 \fi
3582 }
3583 \def\collargs@gobbleOneB@iii{%
3584 \collargs@grabspaces{%
3585 \collargs@letusecollector\collargs@gobbleOneB@iv
3586 }%
3587 }
3588 \def\collargs@gobbleOneB@iv#1{%
3589 \def\collargs@temp{#1}%
3590 \ifx\collargs@temp\collargs@tagarg
```

This is the tag argument we've been waiting for!

```
3591 \else
```

Nope, this `\begin` belongs to someone else.

```
3592 \def\collargs@begins@increment{0}%
3593 \fi
```

Whatever the result was, we have to append the gobbled group to the temporary toks.

```
3594 \etoksapp0{\collargs@grabbed@spaces\unexpanded{{#1}}}%
3595 \collargs@init@grabspaces
3596 \collargs@gobbleOneB@vi
3597 }
3598 \def\collargs@gobbleOneB@v#1{\collargs@gobbleOneB@vi}
3599 \def\collargs@gobbleOneB@vi{%
```

Store.

```
3600 \etoksapp\collargsArg{\the\toks0}%
```

Advance the begin-tag counter.

```
3601 \advance\collargs@begins\collargs@begins@increment\relax
```

Find more begin-tags, unless this was the final one.

```
3602 \ifnum\collargs@begins@increment=-1
3603 \else
3604 \expandafter\collargs@gobbleOneB\expandafter\collargs@empty
3605 \fi
3606 }
3607 \def\collargs@gobbleUntilE@i{%
```

Do we have to check the tag argument (i.e. the environment name after `\end`)?

```
3608 \expandafter\ifx\expandafter\relax\collargs@tagarg\relax
3609 \expandafter\collargs@gobbleUntilE@iv
3610 \else
```

Yup, so let's (carefully) collect the tag argument.

```
3611 \expandafter\collargs@gobbleUntilE@ii
3612 \fi
3613 }
3614 \def\collargs@gobbleUntilE@ii{%
3615 \collargs@grabspaces{%
3616 \collargs@letusecollector\collargs@gobbleUntilE@iii
3617 }%
3618 }
```



```

3619 \def\collargs@gobbleUntilE@iii#1{%
3620 \etoksapp0{\collargs@grabbed@spaces}%
3621 \collargs@init@grabspaces
3622 \def\collargs@tempa{#1}%
3623 \ifx\collargs@tempa\collargs@tagarg

```

This is the tag argument we've been waiting for!

```

3624 \expandafter\collargs@gobbleUntilE@iv
3625 \else

```

Nope, this `\end` belongs to someone else. Insert the end tag plus the tag argument, and collect until the next `\end`.

```

3626 \expandafter\toksapp\expandafter0\expandafter{\collargs@endtag{#1}}%
3627 \expandafter\collargs@letusecollector\expandafter\collargs@gobbleUntilE
3628 \fi
3629 }
3630 \def\collargs@gobbleUntilE@iv{%

```

Invoke `\collargs@gobbleOneB` with the collected material, plus a fake begin-tag and a quark guard.

```

3631 \ifcollargsIgnoreNesting
3632 \expandafter\collargsArg\expandafter{\the\toks0}%
3633 \expandafter\collargs@commandatend
3634 \else
3635 \expandafter\collargs@gobbleUntilE@v
3636 \fi
3637 }
3638 \def\collargs@gobbleUntilE@v{%
3639 \expanded{%
3640 \noexpand\collargs@letusecollector\noexpand\collargs@gobbleOneB
3641 \noexpand\collargs@empty
3642 \the\toks0

```

Add a fake begin-tag and a quark guard.

```

3643 \expandonce\collargs@begintag
3644 \noexpand\collargs@qend
3645 }%
3646 \ifnum\collargs@begins<0
3647 \expandafter\collargs@commandatend
3648 \else
3649 \etoksapp\collargsArg{%
3650 \expandonce\collargs@endtag
3651 \expandafter\ifx\expandafter\relax\collargs@tagarg\relax\else{%
3652 \expandonce\collargs@tagarg}\fi
3653 }%
3654 \toks0={}%
3655 \expandafter\collargs@letusecollector\expandafter\collargs@gobbleUntilE
3656 \expandafter\collargs@empty
3657 \fi
3658 }

```

`\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.

```

3659 \def\collargs@e{%

```

We open an extra group, because `\collargs@appendarg` will close a group for each embellishment.

```

3660 \global\collargs@fix@requestedtrue
3661 \begingroup

```

```

3662 \ifcollargs@verbatim
3663   \expandafter\collargs@e@verbatim
3664 \else
3665   \expandafter\collargs@e@i
3666 \fi
3667 }

```

Detokenize the embellishment tokens in the verbatim mode.

```

3668 \def\collargs@e@verbatim#1{%
3669   \expandafter\collargs@e@i\expandafter{\detokenize{#1}}%
3670 }

```

Ungroup the embellishment tokens, separating them from the rest of the argument specification by a dot.

```

3671 \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.

```

3672 \def\collargs@e@ii#1.#2.{%
3673   \collargs@grabspaces{\collargs@e@iii#1.#2.}%
3674 }

```

What's the argument token?

```

3675 \def\collargs@e@iii#1.#2.{%
3676   \def\collargs@e@iv{\collargs@e@v#1.#2.}%
3677   \futurelet\collargs@temp\collargs@e@iv
3678 }

```

If it is a open or close group character, we surely don't have an embellishment.

```

3679 \def\collargs@e@v{%
3680   \ifcat\noexpand\collargs@temp\bgroup\relax
3681     \let\collargs@marshal\collargs@e@z
3682   \else
3683     \ifcat\noexpand\collargs@temp\egroup\relax
3684       \let\collargs@marshal\collargs@e@z
3685     \else
3686       \let\collargs@marshal\collargs@e@vi
3687     \fi
3688   \fi
3689   \collargs@marshal
3690 }

```

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.

```

3691 \newif\ifcollargs@in@
3692 \def\collargs@in@#1#2{%
3693   \def\collargs@in@@##1##2##3\collargs@in@@{%
3694     \ifx\collargs@in@@##2\global\collargs@in@false\else\global\collargs@in@true\fi
3695   }%
3696   {\collargs@in@@##2#1\collargs@in@\collargs@in@@}%
3697 }

```

Let's see whether the following token, now #3, is an embellishment token.

```

3698 \def\collargs@e@vi#1.#2.#3{%
3699   \collargs@in@{#3}{#1}%
3700   \ifcollargs@in@

```

```

3701 \expandafter\collargs@e@vii
3702 \else
3703 \expandafter\collargs@e@z
3704 \fi
3705 #1.#2.#3%
3706 }

```

#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`.

```
3707 \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.

```

3708 \let\collargs@real@appendarg\collargs@appendarg
3709 \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.

```

3710 \def\collargs@{\collargs@e@viii#1.#3}%
3711 \collargs@m#2.%
3712 }

```

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.

```
3713 \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).

```
3714 \collargs@wrap{#2##1}%
```

We need to get rid of the current embellishment from embellishments, not to catch the same embellishment twice.

```

3715 \def\collargs@e@ix##1#2{\collargs@e@x##1}%
3716 \collargs@e@ix#1.#3.%
3717 }

```

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.

```
3718 \def\collargs@e@x{%
```

Process the argument and append it to the storage.

```
3719 \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.

```

3720 \begingroup
3721 \collargs@e@ii
3722 }

```

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.

```
3723 \def\collargs@e@z#1.{\endgroup\endgroup\collargs@}
```

`\collargs@E` Discard the defaults and execute e.

```
3724 \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.

```
3725 \let\collargs@NoVerbatimAfterNoVerbatim\relax
3726 \def\collargs@VerbAfterNoVerbatim{%
3727   \collargs@verbatimtrue
3728   \collargs@verbatimbracesfalse
3729   \collargs@make@verbatim
3730   \collargs@after{Verb}%
3731 }
3732 \def\collargs@VerbatimAfterNoVerbatim{%
3733   \collargs@verbatimtrue
3734   \collargs@verbatimbracestrue
3735   \collargs@make@verbatim
3736   \collargs@after{Verbatim}%
3737 }
3738 \def\collargs@NoVerbatimAfterVerb{%
3739   \collargs@verbatimfalse
3740   \collargs@verbatimbracesfalse
3741   \collargs@make@other@groups
3742   \collargs@make@no@verbatim
3743   \collargs@after{NoVerbatim}%
3744 }
3745 \def\collargs@VerbAfterVerb{%
3746   \collargs@make@other@groups
3747 }
3748 \def\collargs@VerbatimAfterVerb{%
3749   \collargs@verbatimbracestrue
3750   \collargs@make@other@groups
```

Process the lists of grouping characters, created by `\collargs@make@verbatim`, making these characters of category “other”.

```
3751 \def\collargs@do##1{\catcode##1=12 }%
3752 \collargs@bgroups
3753 \collargs@egroups
3754 \collargs@after{Verbatim}%
3755 }%
3756 \let\collargs@NoVerbatimAfterVerbatim\collargs@NoVerbatimAfterVerb
3757 \def\collargs@VerbAfterVerbatim{%
3758   \collargs@verbatimbracesfalse
3759   \collargs@make@other@groups
```

Process the lists of grouping characters, created by `\collargs@make@verbatim`, making these characters be of their normal category.

```
3760 \def\collargs@do##1{\catcode##1=1 }%
3761 \collargs@bgroups
```

```

3762 \def\collargs@do##1{\catcode##1=2 }%
3763 \collargs@egroups
3764 \collargs@after{Verb}%
3765 }%
3766 \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.

```

3767 \def\collargs@after#1{%
3768 \letcs\collargsVerbatim{collargs@VerbatimAfter#1}%
3769 \letcs\collargsVerb{collargs@VerbAfter#1}%
3770 \letcs\collargsNoVerbatim{collargs@NoVerbatimAfter#1}%
3771 }

```

The first transition is always from the non-verbatim mode.

```

3772 \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.

```

3773 \def\collargs@bgroups{}%
3774 \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.)

```

3775 \ifdefined\luatexversion
3776 \def\collargs@cc#1{%
3777 \directlua{tex.sprint(tex.getcatcode(\collargs@catcodetable@original,
3778 \the\numexpr\expandafter`\csname#1\endcsname\relax))}%
3779 }
3780 \else
3781 \def\collargs@cc#1{%
3782 \ifcsname collargs@cc@\the\numexpr\expandafter`\csname#1\endcsname\endcsname
3783 \csname collargs@cc@\the\numexpr\expandafter`\csname#1\endcsname\endcsname
3784 \else
3785 12%
3786 \fi
3787 }
3788 \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 **\collargs@Braces** 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 **\collargs@Braces** (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 **\collargs@Braces** merely sets **\collargs@make@other@groups**, which gets executed by the subsequent **\collargsVerbatim**, **\collargsVerb** or **\collargsNoVerbatim** (either directly or via **\collargs@make@verbatim**).

```

3789 \def\collargs@Braces#1{%
3790 \expandafter\collargs@braces@i\detokenize{#1}\relax
3791 }
3792 \def\collargs@braces@i#1#2#3\relax{%

```

```

3793 \def\collargs@make@other@groups{%
3794   \def\collargs@other@bgroup{#1}%
3795   \def\collargs@other@egroup{#2}%
3796 }%
3797 }
3798 \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.

```

3799 \ifdefined\luatexversion
3800 (*latex, context)
3801 \newcatcodetable\collargs@catcodetable@verbatim
3802 (latex) \let\collargs@catcodetable@atletter\catcodetable@atletter
3803 (context) \newcatcodetable\collargs@catcodetable@atletter
3804 (/latex, context)
3805 (*plain)
3806 \ifdefined\collargs@catcodetable@verbatim\else
3807   \chardef\collargs@catcodetable@verbatim=4242
3808 \fi
3809 \chardef\collargs@catcodetable@atletter=%
3810   \number\numexpr\collargs@catcodetable@verbatim+1\relax
3811 \chardef\collargs@catcodetable@initex=%
3812   \number\numexpr\collargs@catcodetable@verbatim+2\relax
3813 \initcatcodetable\collargs@catcodetable@initex
3814 (/plain)
3815 (plain, context) \savecatcodetable\collargs@catcodetable@atletter
3816 \begingroup
3817 \@firstofone{%
3818 (latex) \catcodetable\catcodetable@initex
3819 (plain) \catcodetable\collargs@catcodetable@initex
3820 (context) \catcodetable\inicatcodes
3821 \catcode`\=12
3822 \catcode13=12
3823 \catcode0=12
3824 \catcode32=12
3825 \catcode`\%=12
3826 \catcode127=12
3827 \def\collargs@do#1{\catcode#1=12 }%
3828 \collargs@forrange{`a}{`z}%
3829 \collargs@forrange{`A}{`Z}%
3830 \savecatcodetable\collargs@catcodetable@verbatim
3831 \endgroup
3832 }%
3833 \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.

```

3834 \collargsSet{
3835   verbatim ranges/.store in=\collargs@verbatim@ranges,
3836 }
3837 \def\collargsVerbatimRanges#1{\def\collargs@verbatim@ranges{#1}}
3838 \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 Lua<sub>T</sub><sub>E</sub><sub>X</sub>, 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 X<sub>Y</sub><sub>T</sub><sub>E</sub><sub>X</sub>, we perform the change for characters in `verbatim` ranges.

```

3839 \ifdefined\luatexversion
3840   \def\collargs@make@verbatim{%
3841     \directlua{%
3842       for from, to in string.gmatch(
3843         "\luaescapestring{\collargs@verbatim@ranges}",
3844         "(\collargs@percentchar d+)-(\collargs@percentchar d+)"
3845       ) do
3846         for char = tex.round(from), tex.round(to) do
3847           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 Lua<sub>T</sub><sub>E</sub><sub>X</sub>.

```

3848         if catcode == 1 then
3849           tex.sprint(
3850             \number\collargs@catcodetable@atletter,
3851             "\noexpand\collargs@make@verbatim@bgroup{" .. char .. "}")
3852         elseif catcode == 2 then
3853           tex.sprint(
3854             \number\collargs@catcodetable@atletter,
3855             "\noexpand\collargs@make@verbatim@egroup{" .. char .. "}")
3856         elseif catcode == 14 then
3857           tex.sprint(
3858             \number\collargs@catcodetable@atletter,
3859             "\noexpand\collargs@make@verbatim@comment{" .. char .. "}")
3860         end
3861       end
3862     end
3863   }%
3864   \edef\collargs@catcodetable@original{\the\catcodetable}%
3865   \catcodetable\collargs@catcodetable@verbatim

```

Even in Lua<sub>T</sub><sub>E</sub><sub>X</sub>, we switch between the `verbatim` braces regimes by hand.

```

3866   \ifcollargs@verbatimbraces
3867   \else
3868     \def\collargs@do##1{\catcode##1=1\relax}%
3869     \collargs@bgroups
3870     \def\collargs@do##1{\catcode##1=2\relax}%
3871     \collargs@egroups
3872   \fi
3873 }
3874 \else

```

The non-Lua<sub>T</sub><sub>E</sub><sub>X</sub> version:

```

3875 \def\collargs@make@verbatim{%
3876   \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.

```

3877     \collargs@make@other@groups
3878     \def\collargs@make@other@groups{}%
3879     \let\collargs@make@other@bgroup\@gobble
3880     \let\collargs@make@other@egroup\@gobble
3881 }%

```

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).

```

3882     \gdef\collargs@comments{}%
3883     \let\collargs@do\collargs@make@verbatim@char
3884     \expandafter\collargs@forranges\expandafter{\collargs@verbatim@ranges}%
3885 }
3886 \def\collargs@make@verbatim@char#1{%

```

Store the current category code of the current character.

```

3887     \ifnum\catcode#1=12
3888     \else
3889         \csedef{collargs@cc@#1}{\the\catcode#1}%
3890     \fi
3891     \ifnum\catcode#1=1
3892         \collargs@make@verbatim@bgroup{#1}%
3893     \else
3894         \ifnum\catcode#1=2
3895             \collargs@make@verbatim@egroup{#1}%
3896         \else
3897             \ifnum\catcode#1=14
3898                 \collargs@make@verbatim@comment{#1}%
3899             \fi

```

Change the category code of the current character (including the comment characters).

```

3900         \ifnum\catcode#1=12
3901         \else
3902             \catcode#1=12\relax
3903         \fi
3904     \fi
3905 \fi
3906 }
3907 \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`.

```

3908 \def\collargs@make@verbatim@bgroup#1{%
3909     \ifcollargs@verbatimbraces
3910         \catcode#1=12\relax
3911     \fi
3912     \appto\collargs@bgroups{\collargs@do{#1}}%
3913     \collargs@make@other@bgroup{#1}%
3914 }
3915 \def\collargs@make@other@bgroup#1{%
3916     \collargs@make@char\collargs@other@bgroup{#1}{12}%
3917     \let\collargs@make@other@bgroup\@gobble
3918 }

```



`\collargs@make@verbatim@egroup` Ditto for the closing group character.

```
3919 \def\collargs@make@verbatim@egroup#1{%
3920   \ifcollargs@verbatimbraces
3921     \catcode#1=12\relax
3922   \fi
3923   \appto\collargs@egroups{\collargs@do{#1}}%
3924   \collargs@make@other@egroup{#1}%
3925 }
3926 \def\collargs@make@other@egroup#1{%
3927   \collargs@make@char\collargs@other@egroup{#1}{12}%
3928   \let\collargs@make@other@egroup\@gobble
3929 }
```

`\collargs@make@verbatim@comment` This macro populates `\collargs@make@comments@other`.

```
3930 \def\collargs@make@verbatim@comment#1{%
3931   \gappto\collargs@comments{\collargs@do{#1}}%
3932 }
```

`\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.

```
3933 \ifdefined\luatexversion
3934   \def\collargs@make@no@verbatim{%
3935     \catcodetable\collargs@catcodetable@original\relax
3936   }%
3937 \else
3938 \def\collargs@make@no@verbatim{%
3939   \let\collargs@do\collargs@make@no@verbatim@char
3940   \expandafter\collargs@forranges\expandafter{\collargs@verbatim@ranges}%
3941 }
3942 \fi
3943 \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.)

```
3944   \ifcsname collargs@cc@#1\endcsname
3945     \catcode#1=\csname collargs@cc@#1\endcsname\relax
```

We don't have to restore category code 12.

```
3946   \fi
3947 }
```

## 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.

```
3948 \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.

```
3949 \long\def\collargs@fix#1{%
```

Going through `\edef + \unexpanded` avoids doubling the hashes.

```
3950 \edef\collargs@fix@next{\unexpanded{#1}}%
3951 \ifcollargs@fix@requested
3952   \letcs\collargs@action{collargs@fix@%
3953     \ifcollargs@last@verbatim
3954     \ifcollargs@last@verbatimbraces V\else v\fi
3955   \else
3956     N%
3957   \fi
3958   to%
3959   \ifcollargs@verbatim
3960   \ifcollargs@verbatimbraces V\else v\fi
3961   \else
3962     N%
3963   \fi
3964 }%
3965 \else
3966   \let\collargs@action\collargs@fix@next
3967 \fi
3968 \collargs@action
3969 }
```

`\collargs@fix@NtoN` Nothing to do, continue with the next-code.

```
\collargs@fix@vtov
\collargs@fix@VtoV 3970 \def\collargs@fix@NtoN{\collargs@fix@next}
3971 \let\collargs@fix@vtov\collargs@fix@NtoN
3972 \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`.

```
3973 \def\collargs@fix@Ntov{%
3974   \futurelet\collargs@temp\collargs@fix@cc@to@other@ii
3975 }
3976 \def\collargs@fix@cc@to@other@ii{%
3977   \ifcat\noexpand\collargs@temp\bgroup
3978     \let\collargs@action\collargs@fix@next
3979   \else
3980     \ifcat\noexpand\collargs@temp\egroup
3981     \let\collargs@action\collargs@fix@next
3982   \else
3983     \let\collargs@action\collargs@fix@NtoV
3984   \fi
3985 \fi
3986 \collargs@action
3987 }
```

`\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.

```
3988 \def\collargs@fix@NtoV{%
```

```

3989 \ifcollargs@double@fix
3990   \ifcollargs@in@second@fix
3991     \expandafter\expandafter\expandafter\collargs@fix@NtoV@secondfix
3992   \else
3993     \expandafter\expandafter\expandafter\collargs@fix@NtoV@onemore
3994   \fi
3995 \else
3996   \expandafter\collargs@fix@NtoV@singlefix
3997 \fi
3998 }

```

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)

```

3999 \def\collargs@fix@NtoV@singlefix{%
4000   \expandafter\collargs@fix@next\string
4001 }

```

If this is the first fix of two, we know #1 is a control sequence, so it is safe to grab it.

```

4002 \def\collargs@fix@NtoV@onemore#1{%
4003   \collargs@do@one@more@fix{%
4004     \expandafter\collargs@fix@next\string#1%
4005   }%
4006 }

```

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.

```

4007 \def\collargs@fix@NtoV@secondfix{%
4008   \if\noexpand\collargs@temp\relax
4009     \expandafter\collargs@fix@NtoV@secondfix@i
4010   \else
4011     \expandafter\collargs@fix@NtoV@singlefix
4012   \fi
4013 }
4014 \def\collargs@fix@NtoV@secondfix@i#1{%
4015   \gdef\collargs@double@fix@cs@ii{#1}%
4016   \collargs@fix@NtoV@singlefix#1%
4017 }

```

`\collargs@fix@vtoN` Do nothing for the grouping tokens, redirect to `\collargs@fix@VtoN` for other tokens.

```

4018 \def\collargs@fix@vtoN{%
4019   \futurelet\collargs@token\collargs@fix@vtoN@i
4020 }
4021 \def\collargs@fix@vtoN@i{%
4022   \ifcat\noexpand\collargs@token\bgroup
4023     \expandafter\collargs@fix@next
4024   \else
4025     \ifcat\noexpand\collargs@token\egroup
4026     \expandafter\expandafter\expandafter\collargs@fix@next
4027   \else
4028     \expandafter\expandafter\expandafter\collargs@fix@VtoN
4029   \fi
4030 \fi
4031 }

```

`\collargs@fix@vtoV` Redirect group tokens to `\collargs@fix@NtoV`, and do nothing for other tokens.

```

4032 \def\collargs@fix@vtoV{%
4033   \futurelet\collargs@token\collargs@fix@vtoV@i

```

```

4034 }
4035 \def\collargs@fix@vtoV@i{%
4036 \ifcat\noexpand\collargs@token\bgroup
4037 \expandafter\collargs@fix@NtoV
4038 \else
4039 \ifcat\noexpand\collargs@token\egroup
4040 \expandafter\expandafter\expandafter\collargs@fix@NtoV
4041 \else
4042 \expandafter\expandafter\expandafter\collargs@fix@next
4043 \fi
4044 \fi
4045 }

```

`\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.

```

4046 \def\collargs@fix@catcode@of@braces@fromverbatim#1{%
4047 \ifnum\catcode`#1=1
4048 \expandafter\collargs@fix@VtoN
4049 \expandafter#1%
4050 \else
4051 \ifnum\catcode`#1=2
4052 \expandafter\expandafter\expandafter\collargs@fix@cc@VtoN
4053 \expandafter\expandafter\expandafter#1%
4054 \else
4055 \expandafter\expandafter\expandafter\collargs@fix@next
4056 \fi
4057 \fi
4058 }

```

`\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.

```

4059 \def\collargs@fix@VtoN#1{%
4060 \ifnum\catcode`#1=0
4061 \expandafter\collargs@fix@VtoN@escape
4062 \else
4063 \ifnum\catcode`#1=14
4064 \expandafter\expandafter\expandafter\collargs@fix@VtoN@comment
4065 \else
4066 \expandafter\expandafter\expandafter\collargs@fix@VtoN@token
4067 \fi
4068 \fi
4069 #1%
4070 }

```

`\collargs@fix@VtoN@token` We create a new character with the current category code behind the next-code. This works even for grouping characters.

```

4071 \def\collargs@fix@VtoN@token#1{%
4072 \collargs@insert@char\collargs@fix@next{`#1}{\the\catcode`#1}%
4073 }

```

`\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](https://tex.stackexchange.com/a/10454/16819) by Bruno Le Floch.

```

4074 \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.

```

4075 \def#1##1{%
4076   \begingroup%
4077   \escapechar=`\%
4078   \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  $\text{\^M}$  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.

```

4079   \def\collargs@do####1{\catcode###1=12 }%
4080   \collargs@comments
4081   \csname\string#1\endcsname%
4082 }%
4083 \begingroup%
4084 \escapechar=`\%
4085 \lccode`~=\endlinechar%
4086 \lowercase{%
4087   \expandafter\endgroup
4088   \expandafter\def\csname\string#1\endcsname##1~%
4089 }f%

```

I have removed `\space` from the end of the following line. We don’t want it for our application.

```

4090   \endgroup#2%
4091 }%
4092 }
4093 \collargs@defcommentstripper\collargs@fix@VtoN@comment{%
4094   \collargs@fix@next
4095 }

```

We don’t need the generator any more.

```

4096 \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.

```

4097 \def\collargs@fix@VtoN@escape#1{%
4098   \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.

```

4099   \expandafter\collargs@fix@VtoN@escape@d
4100   \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.

```

4101   \expandafter\collargs@fix@VtoN@escape@i
4102   \fi
4103 }
4104 \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.

```

4105   \begingroup

```

```

4106 \catcode`\ =12
4107 \catcode`\{=12
4108 \catcode`\}=12
4109 \catcode`\_ =12
4110 \collargs@fix@VtoN@escape@ii
4111 }

```

The argument is the first character of the control sequence name.

```

4112 \def\collargs@fix@VtoN@escape@ii#1{%
4113 \endgroup
4114 \def\collargs@csname{#1}%

```

Only if #1 is a letter may the control sequence name continue.

```

4115 \ifnum\catcode`#1=11
4116 \expandafter\collargs@fix@VtoN@escape@iii
4117 \else

```

In the case of a control space, we have to throw away the following spaces.

```

4118 \ifnum\catcode`#1=10
4119 \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@s
4120 \else

```

We have a control symbol. That means that we haven't peeked ahead and can thus skip `\collargs@fix@VtoN@escape@z`.

```

4121 \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@z@i
4122 \fi
4123 \fi
4124 }

```

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).

```

4125 \def\collargs@fix@VtoN@escape@iii{%
4126 \futurelet\collargs@temp\collargs@fix@VtoN@escape@iv
4127 }
4128 \def\collargs@fix@VtoN@escape@iv{%
4129 \ifcat\noexpand\collargs@temp\bgroup
4130 \let\collargs@action\collargs@fix@VtoN@escape@z
4131 \else
4132 \ifcat\noexpand\collargs@temp\egroup
4133 \let\collargs@action\collargs@fix@VtoN@escape@z
4134 \else
4135 \expandafter\ifx\space\collargs@temp
4136 \let\collargs@action\collargs@fix@VtoN@escape@s
4137 \else
4138 \let\collargs@action\collargs@fix@VtoN@escape@v
4139 \fi
4140 \fi
4141 \fi
4142 \collargs@action
4143 }

```

If we have a letter, store it and loop back, otherwise finish.

```

4144 \def\collargs@fix@VtoN@escape@v#1{%
4145 \ifcat\noexpand#1a%
4146 \appto\collargs@csname{#1}%
4147 \expandafter\collargs@fix@VtoN@escape@iii
4148 \else
4149 \expandafter\collargs@fix@VtoN@escape@z\expandafter#1%
4150 \fi
4151 }

```

Throw away the following spaces.

```
4152 \def\collargs@fix@VtoN@escape@s{%
4153   \futurelet\collargs@temp\collargs@fix@VtoN@escape@s@i
4154 }
4155 \def\collargs@fix@VtoN@escape@s@i{%
4156   \expandafter\ifx\space\collargs@temp
4157   \expandafter\collargs@fix@VtoN@escape@s@ii
4158   \else
4159   \expandafter\collargs@fix@VtoN@escape@z
4160   \fi
4161 }
4162 \def\collargs@fix@VtoN@escape@s@ii{%
4163   \expandafter\collargs@fix@VtoN@escape@z\romannumeral-0%
4164 }
```

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.

```
4165 \def\collargs@fix@VtoN@escape@z@i{%
4166   \collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin
4167   \collargs@fix@VtoN@escape@z@ii
4168 }%
4169 \def\collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin{%
4170   \ifcsname\collargs@csname\endcsname
4171     \@tempwattrue
4172   \else
4173     \@tempwafalse
4174   \fi
4175 }
4176 \def\collargs@fix@VtoN@escape@z@maybe@undefine@cs@end{%
4177   \if@tempswa
4178   \else
4179     \cslet{\collargs@csname}\collargs@undefined
4180   \fi
4181 }
4182 \def\collargs@fix@VtoN@escape@z@ii{%
4183   \expandafter\collargs@fix@VtoN@escape@z@maybe@undefine@cs@end
4184   \expandafter\collargs@fix@next\csname\collargs@csname\endcsname
4185 }
```

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.

```
4186 \def\collargs@fix@VtoN@escape@z{%
4187   \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.

```
4188   \collargs@do@one@more@fix\collargs@fix@VtoN@escape@z@i
4189 }-%
```

Remember the collected control sequence. It will be used in `\collargs@cancel@double@fix`.

```
4190 \collargs@fix@VtoN@escape@z@maybe@undefine@cs@begin
4191 \xdef\collargs@double@fix@cs@i{\expandonce{\csname\collargs@csname\endcsname}}%
4192 \collargs@fix@VtoN@escape@z@maybe@undefine@cs@end
```

Request the double-fix.

```
4193 \global\collargs@double@fixtrue
```

The complication is addressed, redirect to the control symbol finish.

```
4194 \collargs@fix@VtoN@escape@z@ii
4195 }%
4196 }
```

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`.

```
4197 \def\collargs@fix@VtoN@escape@d{%
4198 \ifcollargs@in@second@fix
4199 \expandafter\collargs@fix@VtoN@escape@d@i
4200 \expandafter\collargs@double@fix@cs@ii
4201 \else
4202 \expandafter\collargs@fix@VtoN@escape@d@i
4203 \expandafter\collargs@double@fix@cs@i
4204 \fi
4205 }
```

We have the contents of either `\collargs@double@fix@cs@i` or `\collargs@double@fix@cs@ii` here, a control sequence in both cases.

```
4206 \def\collargs@fix@VtoN@escape@d@i#1{%
4207 \expandafter\expandafter\expandafter\collargs@fix@VtoN@escape@d@ii
4208 \expandafter\string#1\relax
4209 }
```

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`.

```
4210 \def\collargs@fix@VtoN@escape@d@ii#1#2\relax{%
4211 \def\collargs@csname{#2}%
4212 \def\collargs@fix@VtoN@escape@d@iii#2{%
4213 \collargs@fix@VtoN@escape@z
4214 }%
4215 \collargs@fix@VtoN@escape@d@iii
4216 }
```

This conditional signals a double-fix request. It should be always set globally, because it is cleared by `\collargs@double@fixfalse` in a group.

```
4217 \newif\ifcollargs@double@fix
```

This conditional signals that we’re currently performing the second fix.

```
4218 \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.

```

4219 \def\collargs@if@one@more@fix{%
4220   \ifcollargs@double@fix
4221     \ifcollargs@in@second@fix
4222       \expandafter\expandafter\expandafter\@secondoftwo
4223     \else
4224       \expandafter\expandafter\expandafter\@firstoftwo
4225     \fi
4226   \else
4227     \expandafter\@secondoftwo
4228   \fi
4229 }
4230 \def\collargs@do@one@more@fix#1{%

```

We perform the second fix in a group, signalling that we’re performing it.

```

4231   \begingroup
4232   \collargs@in@second@fixtrue

```

Reexecute the fixing routine, at the end, close the group and execute the given code afterwards.

```

4233   \collargs@fix{%
4234     \endgroup
4235     #1%
4236   }%
4237 }

```

This macro is called from `\collargs@appendarg` to cancel the double-fix request.

```

4238 \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.

```

4239   \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”.

```

4240   \begingroup

```

This will attach the delimiters directly to the argument, so we’ll see what was actually consumed.

```

4241   \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.

```

4242   \edef\collargs@temp{\the\collargsArg}%
4243   \edef\collargs@tempa{\expandafter\string\collargs@double@fix@cs@i}%
4244   \ifx\collargs@temp\collargs@tempa
4245     \global\collargs@fix@requestedtrue
4246   \fi
4247   \endgroup
4248 }

```

`\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 $\TeX$ , Xe $\TeX$  and L $\TeX$  where possible. In the end, we only have to implement our own macros for plain pdf $\TeX$ .

```

4249 (!context) \ifdefined\luatexversion
4250   \def\collargs@insert@char#1#2#3{%
4251     \edef\collargs@temp{\unexpanded{#1}}%
4252     \expandafter\collargs@temp\directlua{%
4253       tex.cprint(\number#3,string.char(\number#2))}%
4254   }%
4255   \def\collargs@make@char#1#2#3{%
4256     \edef#1{\directlua{tex.cprint(\number#3,string.char(\number#2))}}%
4257   }%
4258 (*!context)
4259 \else
4260   \ifdefined\XeTeXversion
4261     \def\collargs@insert@char#1#2#3{%
4262       \edef\collargs@temp{\unexpanded{#1}}%
4263       \expandafter\collargs@temp\Ucharcat #2 #3
4264     }%
4265     \def\collargs@make@char#1#2#3{%
4266       \edef#1{\Ucharcat#2 #3}%
4267     }%
4268   \else
4269     (*!latex)
4270     \ExplSyntaxOn
4271     \def\collargs@insert@char#1#2#3{%
4272       \edef\collargs@temp{\unexpanded{#1}}%
4273       \expandafter\expandafter\expandafter\collargs@temp\char_generate:nn{#2}{#3}%
4274     }%
4275     \def\collargs@make@char#1#2#3{%
4276       \edef#1{\char_generate:nn{#2}{#3}}%
4277     }%
4278     \ExplSyntaxOff
4279     (/!latex)
4280     (*!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.

```

4281   \begingroup
4282   \catcode`^^@=1 \csgdef{collargs@charofcat@1}{%
4283     \noexpand\expandafter^^@\iffalse}\fi}
4284   \catcode`^^@=2 \csgdef{collargs@charofcat@2}{\iffalse{\fi^^@}}
4285   \catcode`^^@=3 \csgdef{collargs@charofcat@3}{^^@}
4286   \catcode`^^@=4 \csgdef{collargs@charofcat@4}{^^@}

```

As we have grabbed the spaces already, a remaining newline should surely be fixed into a `\par`.

```

4287           \csgdef{collargs@charofcat@5}{\par}
4288   \catcode`^^@=6 \csxdef{collargs@charofcat@6}{\unexpanded{^^@}}
4289   \catcode`^^@=7 \csgdef{collargs@charofcat@7}{^^@}
4290   \catcode`^^@=8 \csgdef{collargs@charofcat@8}{^^@}
4291           \csgdef{collargs@charofcat@10}{\noexpand\space}
4292   \catcode`^^@=11 \csgdef{collargs@charofcat@11}{^^@}
4293   \catcode`^^@=12 \csgdef{collargs@charofcat@12}{^^@}
4294   \catcode`^^@=13 \csgdef{collargs@charofcat@13}{^^@}
4295   \endgroup
4296   \def\collargs@insert@char#1#2#3{%

```

Temporarily change the lowercase code of  $\text{\char"}$  to the requested character #2.

```
4297 \begingroup
4298 \lccode\char"#2\relax
```

We'll have to close the group before executing the next-code.

```
4299 \def\collargs@temp{\endgroup#1}%
```

$\text{\collargs@charofcat}\langle\textit{requested category code}\rangle$  is f-expanded first, leaving us to lowercase  $\text{\expandafter\collargs@temp}\text{\char"}$ . Clearly, lowercasing  $\text{\expandafter\collargs@temp}$  is a no-op, but lowercasing  $\text{\char"}$  gets us the requested character of the requested category.  $\text{\expandafter}$  is executed next, and this gets rid of the conditional for category codes 1 and 2.

```
4300 \expandafter\lowercase\expandafter{%
4301 \expandafter\expandafter\expandafter\collargs@temp
4302 \romannumeral-`0\csname collargs@charofcat@the\numexpr#3\relax\endcsname
4303 }%
4304 }
```

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.

```
4305 \def\collargs@make@char#1#2#3{%
4306 \begingroup
4307 \lccode\char"#2\relax
```

Define  $\text{\collargs@temp}$  to hold  $\text{\char"}$  of the appropriate category.

```
4308 \edef\collargs@temp{%
4309 \csname collargs@charofcat@the\numexpr#3\relax\endcsname}%
```

Preexpand the second  $\text{\collargs@temp}$  so that we lowercase  $\text{\def\collargs@temp}\text{\char"}$ , with  $\text{\char"}$  of the appropriate category.

```
4310 \expandafter\lowercase\expandafter{%
4311 \expandafter\def\expandafter\collargs@temp\expandafter{\collargs@temp}%
4312 }%
4313 \expandafter\endgroup
4314 \expandafter\def\expandafter#1\expandafter{\collargs@temp}%
4315 }
4316 ⟨/plain⟩
4317 \fi
4318 \fi
4319 ⟨/!context⟩
```

```
4320 ⟨plain⟩\resetatcatcode
4321 ⟨context⟩\stopmodule
4322 ⟨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`

```
4323 my $PROG = 'memoize-extract.pl';
4324 my $VERSION = '2024/01/21 v1.1.2';
4325
4326 use strict;
4327 use File::Basename qw/basename/;
4328 use Getopt::Long;
```

```

4329 use File::Spec::Functions
4330     qw/splitpath catpath splitdir rootdir file_name_is_absolute/;
4331 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.

```

4332 our ($pdf_file, $prune, $keep, $format, $force, $quiet,
4333     $pdf_library, $print_version, $mkdir, $help,
4334     $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.

```

4335 my %ERROR = (
4336     latex => '\PackageError{memoize (perl-based extraction)}{${short}}{${long}}',
4337     plain => '\errhelp{${long}}\errmsg{memoize (perl-based extraction): ${short}}',
4338     context => '\errhelp{${long}}\errmsg{memoize (perl-based extraction): ${short}}',
4339     '' => '$header${short}. ${long}');
4340
4341 my %WARNING = (
4342     latex => '\PackageWarning{memoize (perl-based extraction)}{${texindent}text}',
4343     plain => '\message{memoize (perl-based extraction) Warning: ${texindent}text}',
4344     context => '\message{memoize (perl-based extraction) Warning: ${texindent}text}',
4345     '' => '$header${indent}text. ');
4346
4347 my %INFO = (
4348     latex => '\PackageInfo{memoize (perl-based extraction)}{${texindent}text}',
4349     plain => '\message{memoize (perl-based extraction): ${texindent}text}',
4350     context => '\message{memoize (perl-based extraction): ${texindent}text}',
4351     '' => '$header${indent}text. ');

```

Some variables used in the message routines; note that `header` will be redefined once we parse the arguments.

```

4352 my $exit_code = 0;
4353 my $log;
4354 my $header = '';
4355 my $indent = '';
4356 my $texindent = '';

```

The message routines.

```

4357 sub error {
4358     my ($short, $long) = @_;
4359     if (! $quiet) {
4360         $_ = $ERROR{' '};
4361         s/\$header/\$header/;
4362         s/\$short/\$short/;
4363         s/\$long/\$long/;
4364         print(STDOUT "$_\n");
4365     }
4366     if ($log) {
4367         $long =~ s/\\/\\/string\\/g;
4368         $_ = $ERROR{$format};
4369         s/\$short/\$short/;
4370         s/\$long/\$long/;
4371         print(LOG "$_\n");
4372     }
4373     $exit_code = 11;
4374     endinput();
4375 }
4376

```

```

4377 sub warning {
4378     my $text = shift;
4379     if ($log) {
4380         $_ = $WARNING{$format};
4381         s/\$texindent/$texindent/;
4382         s/\$text/$text/;
4383         print(LOG "$_\n");
4384     }
4385     if (! $quiet) {
4386         $_ = $WARNING{' '};
4387         s/\$header/$header/;
4388         s/\$indent/$indent/;
4389         s/\$text/$text/;
4390         print(STDOUT "$_\n");
4391     }
4392     $exit_code = 10;
4393 }
4394
4395 sub info {
4396     my $text = shift;
4397     if ($text && ! $quiet) {
4398         $_ = $INFO{' '};
4399         s/\$header/$header/;
4400         s/\$indent/$indent/;
4401         s/\$text/$text/;
4402         print(STDOUT "$_\n");
4403         if ($log) {
4404             $_ = $INFO{$format};
4405             s/\$texindent/$texindent/;
4406             s/\$text/$text/;
4407             print(LOG "$_\n");
4408         }
4409     }
4410 }

```

Mark the log as complete and exit.

```

4411 sub endinput {
4412     if ($log) {
4413         print(LOG "\\endinput\n");
4414         close(LOG);
4415     }
4416     exit $exit_code;
4417 }
4418
4419 sub die_handler {
4420     stderr_to_warning();
4421     my $text = shift;
4422     chomp($text);
4423     error("Perl error: $text", ' ');
4424 }
4425
4426 sub warn_handler {
4427     my $text = shift;
4428     chomp($text);
4429     warning("Perl warning: $text");
4430 }

```

This is used to print warning messages from PDF::Builder, which are output to STDERR.

```

4431 my $stderr;
4432 sub stderr_to_warning {
4433     if ($stderr) {
4434         my $w = ' Perl info: ';
4435         my $nl = ' ';

```

```

4436     for (split(/\n/, $stderr)) {
4437         /(^\\s*)(.*?)(\\s*)$/;
4438         $w .= ($1 ? ' ' : $nl) . $2;
4439         $nl = "\n";
4440     }
4441     warning("$w");
4442     $stderr = '';
4443 }
4444 }

```

[Permission-related functions](#) We will need these variables below. Note that we only support Unix and Windows.

```

4445 my $on_windows = $^O eq 'MSWin32';
4446 my $dirsep = $on_windows ? '\\\ ' : '/\ ' ;

```

paranoia\_in/out should work exactly as `kpsewhich -safe-in-name/-safe-out-name`.

```

4447 sub paranoia_in {
4448     my ($f, $remark) = @_ ;
4449     error("I'm not allowed to read from '$f' (openin_any = $openin_any)",
4450         $remark) unless _paranoia($f, $openin_any);
4451 }
4452
4453 sub paranoia_out {
4454     my ($f, $remark) = @_ ;
4455     error("I'm not allowed to write to '$f' (openin_any = $openout_any)",
4456         $remark) unless _paranoia($f, $openout_any);
4457 }
4458
4459 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`.

```

4460     my ($f, $mode) = @_ ;
4461     return if (! $f);

```

We split the filename into the directory and the basename part, and the directory into components.

```

4462     my ($volume, $dir, $basename) = splitpath($f);
4463     my @dir = splitdir($dir);
4464     return (

```

In mode 'any' (a, y or 1), we may access any file.

```

4465         $mode =~ /^[ay1]$/
4466         || (

```

Otherwise, we are at least in the restricted mode, so we should not open dot files on Unix-like systems (except file called `.tex`).

```

4467             !( !$on_windows && $basename =~ /\.\ / && !($basename =~ /\.\.tex$/) )
4468             && (

```

If we are precisely in the restricted mode (r, n, 0), then there are no further restrictions.

```

4469                 $mode =~ /^[rn0]$/

```

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.

```

4470                 || (

```

We're not allowed to go to a parent directory.

```

4471                     ! grep(/^\.\./, @dir) && $basename ne '...'
4472                     &&

```

If the given path is absolute, it should be a descendant of either `TEXMF_OUTPUT_DIRECTORY` or `TEXMFOUTPUT`.

```

4473                         (!file_name_is_absolute($f)
4474                         ||

```

```

4475             is_ancestor($texmf_output_directory, $f)
4476             ||
4477             is_ancestor($texmfoutput, $f)
4478         ))));
4479 }

```

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.

```

4480 sub normalize_path {
4481     my $path = shift;
4482     my ($v, $d, $n) = splitpath($path);
4483     if ($n eq '' && $d =~ /^[^Q$dirsep\E]\Q$dirsep\E+$/ ) {
4484         $path =~ s/\Q$dirsep\E+$/ /;
4485     }
4486     return $path;
4487 }

```

On Windows, we disallow “semi-absolute” paths, i.e. paths starting with the `\` but lacking the drive. `File::Spec`'s function `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.

```

4488 sub sanitize_path {
4489     my $f = normalize_path(shift);
4490     my ($v, $d, $n) = splitpath($f);
4491     if ($on_windows) {
4492         my $a = file_name_is_absolute($f);
4493         if ($a == 1 || ($a == 0 && $v) ) {
4494             error("\Semi-absolute" paths are disallowed: " . $f,
4495                 "The path must either both contain the drive letter and " .
4496                 "start with '\\', or none of these; paths like 'C:foo\\bar' " .
4497                 "and '\\foo\\bar' are disallowed");
4498         }
4499     }
4500 }
4501
4502 sub access_in {
4503     return -r shift;
4504 }
4505
4506 sub access_out {
4507     my $f = shift;
4508     my $exists;
4509     eval { $exists = -e $f };

```

Presumably, we get this error when the parent directory is not executable.

```

4510     return if ($@);
4511     if ($exists) {

```

An existing file should be writable, and if it's a directory, it should also be executable.

```

4512         my $rw = -w $f; my $rd = -d $f; my $rx = -x $f;
4513         return -w $f && (! -d $f || -x $f);
4514     } 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.)

```

4515         my $p = parent($f);
4516         return -w $p;
4517     }
4518 }

```

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.

```

4519 sub find_in {
4520     my $f = shift;
4521     sanitize_path($f);
4522     return $f if file_name_is_absolute($f);
4523     for my $df (
4524         $texmf_output_directory ? join_paths($texmf_output_directory, $f) : undef,
4525         $f,
4526         $texmfoutput ? join_paths($texmfoutput, $f) : undef) {
4527         return $df if $df && -r $df;
4528     }
4529     return $f;
4530 }

```

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.

```

4531 sub find_out {
4532     my $f = shift;
4533     sanitize_path($f);
4534     return $f if file_name_is_absolute($f);
4535     for my $df (
4536         $texmf_output_directory ? join_paths($texmf_output_directory, $f) : undef,
4537         $texmf_output_directory ? undef : $f,
4538         $texmfoutput ? join_paths($texmfoutput, $f) : undef) {
4539         return $df if $df && access_out($df);
4540     }
4541     return $texmf_output_directory ? join_paths($texmf_output_directory, $f) : $f;
4542 }

```

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.

```

4543 sub name {
4544     my $path = shift;
4545     my ($volume, $dir, $filename) = splitpath($path);
4546     return $filename;
4547 }
4548
4549 sub suffix {
4550     my $path = shift;
4551     my ($volume, $dir, $filename) = splitpath($path);
4552     $filename =~ /\.[^.]*/;
4553     return $&;
4554 }
4555
4556 sub with_suffix {
4557     my ($path, $suffix) = @_ ;
4558     my ($volume, $dir, $filename) = splitpath($path);
4559     if ($filename =~ s/\.[^.]*/$suffix/) {
4560         return catpath($volume, $dir, $filename);
4561     } else {
4562         return catpath($volume, $dir, $filename . $suffix);
4563     }
4564 }
4565
4566 sub with_name {

```



```

4567     my ($path, $name) = @_;
4568     my ($volume, $dir, $filename) = splitpath($path);
4569     my ($v,$d,$f) = splitpath($name);
4570     die "Runtime error in with_name: " .
4571 "'$name' should not contain the directory component"
4572     unless $v eq '' && $d eq '' && $f eq $name;
4573     return catpath($volume, $dir, $name);
4574 }
4575
4576 sub join_paths {
4577     my $path1 = normalize_path(shift);
4578     my $path2 = normalize_path(shift);
4579     return $path2 if !$path1 || file_name_is_absolute($path2);
4580     my ($volume1, $dir1, $filename1) = splitpath($path1, 'no_file');
4581     my ($volume2, $dir2, $filename2) = splitpath($path2);
4582     die if $volume2;
4583     return catpath($volume1,
4584                   join($dirsep, ($dir1 eq $dirsep ? '' : $dir1, $dir2)),
4585                   $filename2);
4586 }

```

The logical parent. The same as `pathlib.parent` in Python.

```

4587 sub parent {
4588     my $f = normalize_path(shift);
4589     my ($v, $dn, $dummy) = splitpath($f, 1);
4590     my $p_dn = $dn =~ s/[^\Q$dirsep\E]+$/r;
4591     if ($p_dn eq '') {
4592         $p_dn = $dn =~ /^^\Q$dirsep\E/ ? $dirsep : '.';
4593     }
4594     my $p = catpath($v, $p_dn, '');
4595     $p = normalize_path($p);
4596     return $p;
4597 }

```

This function assumes that both paths are absolute; ancestor may be "", signaling a non-path.

```

4598 sub is_ancestor {
4599     my $ancestor = normalize_path(shift);
4600     my $descendant = normalize_path(shift);
4601     return if !$ancestor;
4602     $ancestor .= $dirsep unless $ancestor =~ /\Q$dirsep\E$/;
4603     return $descendant =~ /^^\Q$ancestor/;
4604 }

```

A paranoid `Path.mkdir`. The given folder is preprocessed by `find_out`.

```

4605 sub make_directory {
4606     my $folder = find_out(shift);
4607     if (! -d $folder) {
4608         paranoia_out($folder);

```

Using `make_path` is fine because we know that `TEXMF_OUTPUT_DIRECTORY/TEXMFOUTPUT`, if given, exists, and that "folder" contains no ...

```

4609         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.

```

4610         info("Created directory $folder");
4611     }
4612 }
4613
4614 sub unquote {
4615     shift =~ s/"(.*)" /\1/rg;
4616 }

```

[Kpathsea](#) Get the values of `openin_any`, `openout_any`, `TEXMFOUTPUT` and `TEXMF_OUTPUT_DIRECTORY`.

```
4617 my $maybe_backslash = $on_windows ? ' ' : '\\';
4618 my $query = 'kpsewhich -expand-var=' .
4619     "openin_any=$maybe_backslash\$openin_any," .
4620     "openout_any=$maybe_backslash\$openout_any," .
4621     "TEXMFOUTPUT=$maybe_backslash\$TEXMFOUTPUT";
4622 my $kpsewhich_output = ` $query `;
4623 if (! $kpsewhich_output) {
```

No TeX? (Note that `kpsewhich` should exist in MiKTeX as well.) In absence of `kpathsea` information, we get very paranoid.

```
4624     ($openin_any, $openout_any) = ('p', 'p');
4625     ($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.

```
4626     warning('I failed to execute "kpsewhich", is there no TeX system installed? ' .
4627         'Assuming openin_any = openout_any = "p" ' .
4628         '(i.e. restricting all file operations to non-hidden files ' .
4629         'in the current directory of its subdirectories).');
4630 } else {
4631     $kpsewhich_output =~ /^openin_any=(.*/),openout_any=(.*/),TEXMFOUTPUT=(.*/);
4632     ($openin_any, $openout_any, $texmfoutput) = @{^CAPTURE};
4633     $texmf_output_directory = $ENV{'TEXMF_OUTPUT_DIRECTORY'};
4634     if ($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`.

```
4635     $query = 'initexmf --show-config-value=[Core]AllowUnsafeInputFiles ' .
4636         '--show-config-value=[Core]AllowUnsafeOutputFiles';
4637     my $initexmf_output = ` $query `;
4638     $initexmf_output =~ /^(.*)\n(.*)\n/m;
4639     $openin_any = $1 eq 'true' ? 'a' : 'p';
4640     $openout_any = $2 eq 'true' ? 'a' : 'p';
4641     $texmfoutput = '';
4642     $texmf_output_directory = '';
4643 }
4644 }
```

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`.

```
4645 sub sanitize_output_dir {
4646     return unless my $d = shift;
4647     sanitize_path($d);
```

On Windows, `rootdir` returns `\`, so it cannot possibly match `$d`.

```
4648     return $d if -d $d && $d ne rootdir();
4649 }
4650
4651 $texmfoutput = sanitize_output_dir($texmfoutput);
4652 $texmf_output_directory = sanitize_output_dir($texmf_output_directory);
```

We don't delve into the real script when loaded from the testing code.

```
4653 return 1 if caller;
```

## Arguments

```
4654 my $usage = "usage: $PROG [-h] [-P PDF] [-p] [-k] [-F {latex,plain,context}] [-f] " .
4655     "[-L {PDF::API2,PDF::Builder}] [-q] [-m] [-V] mmz\n";
4656 my $Help = <<END;
4657 Extract extern pages produced by package Memoize out of the document PDF.
4658
```

```

4659 positional arguments:
4660 mmz                the record file produced by Memoize:
4661                    doc.mmz when compiling doc.tex
4662                    (doc and doc.tex are accepted as well)
4663
4664 options:
4665 -h, --help          show this help message and exit
4666 -P PDF, --pdf PDF   extract from file PDF
4667 -p, --prune         remove the extern pages after extraction
4668 -k, --keep          do not mark externs as extracted
4669 -F, --format {latex,plain,context}
4670                    the format of the TeX document invoking extraction
4671 -f, --force         extract even if the size-check fails
4672 -q, --quiet         describe what's happening
4673 -L, --library {PDF::API2, PDF::Builder}
4674                    which PDF library to use for extraction (default: PDF::API2)
4675 -m, --mkdir         create a directory (and exit);
4676                    mmz argument is interpreted as directory name
4677 -V, --version       show program's version number and exit
4678
4679 For details, see the man page or the Memoize documentation.
4680 END
4681
4682 my @valid_libraries = ('PDF::API2', 'PDF::Builder');
4683 Getopt::Long::Configure ("bundling");
4684 GetOptions(
4685     "pdf|P=s"    => \$pdf_file,
4686     "prune|p"    => \$prune,
4687     "keep|k"     => \$keep,
4688     "format|F=s" => \$format,
4689     "force|f"    => \$force,
4690     "quiet|q"    => \$quiet,
4691     "library|L=s" => \$pdf_library,
4692     "mkdir|m"    => \$mkdir,
4693     "version|V"  => \$print_version,
4694     "help|h|?"  => \$help,
4695 ) or die $usage;
4696
4697 if ($help) {print("$usage\n$Help"); exit 0}
4698
4699 if ($print_version) { print("$PROG of Memoize $VERSION\n"); exit 0 }
4700
4701 die "${usage}$PROG: error: the following arguments are required: mmz\n"
4702     unless @ARGV == 1;
4703
4704 die "${usage}$PROG: error: argument -F/--format: invalid choice: '$format' " .
4705     "(choose from 'latex', 'plain', 'context')\n"
4706     unless grep $_ eq $format, ('', 'latex', 'plain', 'context');
4707
4708 die "${usage}$PROG: error: argument -L/--library: invalid choice: '$pdf_library' " .
4709     "(choose from " . join(", ", @valid_libraries) . ")\n"
4710     if $pdf_library && ! grep $_ eq $pdf_library, @valid_libraries;
4711
4712 $header = $format ? basename($0) . ': ' : '';

```

start a new line in the TeX terminal output

```

4713 print("\n") if $format;

```

**Initialization** With `--mkdir`, argument `mmz` is interpreted as the directory to create.

```

4714 if ($mkdir) {
4715     make_directory($ARGV[0]);
4716     exit 0;
4717 }

```

Normalize the mmz argument into a .mmz filename.

```
4718 my $mmz_file = $ARGV[0];
4719 $mmz_file = with_suffix($mmz_file, '.mmz')
4720     if suffix($mmz_file) eq '.tex';
4721 $mmz_file = with_name($mmz_file, name($mmz_file) . '.mmz')
4722     if suffix($mmz_file) ne '.mmz';
```

Once we have the .mmz filename, we can open the log.

```
4723 if ($format) {
4724     my $_log = find_out(with_suffix($mmz_file, '.mmz.log'));
4725     paranoia_out($_log);
4726     info("Logging to '$_log'");
4727     $log = $_log;
4728     open LOG, ">$_log";
4729 }
```

Now that we have opened the log file, we can try loading the PDF processing library.

```
4730 if ($pdf_library) {
4731     eval "use $pdf_library";
4732     error("Perl module '$pdf_library' was not found",
4733         'Have you followed the instructions in section 1.1 of the manual?')
4734     if ($@);
4735 } else {
4736     for (@valid_libraries) {
4737         eval "use $_";
4738         if (!$@) {
4739             $pdf_library = $_;
4740             last;
4741         }
4742     }
4743     if (!$pdf_library) {
4744         error("No suitable Perl module for PDF processing was found, options are " .
4745             join(", ", @valid_libraries),
4746             'Have you followed the instructions in section 1.1 of the manual?');
4747     }
4748 }
```

Catch any errors in the script and output them to the log.

```
4749 $SIG{__DIE__} = \&die_handler;
4750 $SIG{__WARN__} = \&warn_handler;
4751 close(STDERR);
4752 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.

```
4753 my $given_mmz_file = $mmz_file;
4754 $mmz_file = find_in($mmz_file, 1);
4755 if (! -e $mmz_file) {
4756     info("File '$given_mmz_file' does not exist, assuming there's nothing to do");
4757     endinput();
4758 }
4759 paranoia_in($mmz_file);
4760 paranoia_out($mmz_file,
4761     'I would have to rewrite this file unless option --keep is given.')
4762     unless $keep;
```

Determine the PDF filename: it is either given via --pdf, or constructed from the .mmz filename.

```
4763 $pdf_file = with_suffix($given_mmz_file, '.pdf') if !$pdf_file;
4764 $pdf_file = find_in($pdf_file);
4765 paranoia_in($pdf_file);
4766 paranoia_out($pdf_file,
4767     'I would have to rewrite this file because option --prune was given.')
4768     if $prune;
```

Various initializations.

```
4769 my $pdf;
4770 my %extern_pages;
4771 my $new_mmz;
4772 my $tolerance = 0.01;
4773 info("Extracting new externs listed in '$mmz_file' " .
4774      "from '$pdf_file' using Perl module $pdf_library");
4775 my $done_message = "Done (there was nothing to extract)";
4776 $indent = ' ';
4777 $texindent = '\space\space ';
4778 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.

```
4779 open (MMZ, $mmz_file);
4780 while (<MMZ>) {
4781     my $mmz_line = $_;
4782     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.

```
4783         my $prefix = unquote(${prefix});
4784         warning("Cannot parse line '$mmz_line'") unless
4785             $prefix =~ /(?P<dir_prefix>.*\/)?(?P<name_prefix>.*?)/;
4786         $dir_to_make = ${dir_prefix};
4787     } elsif (/^\mmzNewExtern\ *{(?P<extern_path>.*?)}{(?P<page_n>[0-9]+)}#
4788             {(?P<expected_width>[0-9.]*pt){(?P<expected_height>[0-9.]*pt)/x} {
```

Found \mmzNewExtern: extract the extern page into an extern file.

```
4789         $done_message = "Done";
4790         my $ok = 1;
4791         my %m_ne = %+;
```

The extern filename, as specified in .mmz:

```
4792     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.

```
4793     warning("Cannot parse line '$mmz_line'") unless
4794         $extern_file =~ /(?P<dir_prefix>.*\/)?(?P<name_prefix>.*?)#
4795             (?P<code_md5sum>[0-9A-F]{32})-#
4796             (?P<context_md5sum>[0-9A-F]{32})(?:-[0-9]+)?.pdf/x;
```

The actual extern filename:

```
4797     my $extern_file_out = find_out($extern_file);
4798     paranoia_out($extern_file_out);
4799     my $page = ${m_ne}{page_n};
```

Check whether c-memo and cc-memo exist (in any input directory).

```
4800     my $c_memo = with_name($extern_file,
4801                           ${name_prefix} . ${code_md5sum} . '.memo');
4802     my $cc_memo = with_name($extern_file,
4803                           ${name_prefix} . ${code_md5sum} .
4804                           '-' . ${context_md5sum} . '.memo');
4805     my $c_memo_in = find_in($c_memo);
4806     my $cc_memo_in = find_in($cc_memo);
4807     if ((! access_in($c_memo_in) || ! access_in($cc_memo_in)) && !$force) {
4808         warning("I refuse to extract page $page into extern '$extern_file', " .
4809               "because the associated c-memo '$c_memo' and/or " .
4810               "cc-memo '$cc_memo' does not exist");
4811         $ok = '';
4812     }
```

Load the PDF. We only do this now so that we don't load it if there is nothing to extract.

```

4813     if ($ok && ! $pdf) {
4814         if (!access_in($pdf_file)) {
4815             warning("Cannot open '$pdf_file'", '');
4816             endinput();
4817         }

```

Temporarily disable error handling, so that we can catch the error ourselves.

```

4818     $SIG{__DIE__} = undef; $SIG{__WARN__} = undef;

```

All safe, paranoia\_in was already called above.

```

4819     eval { $pdf = $pdf_library->open($pdf_file, msgver => 0) };
4820     $SIG{__DIE__} = \&die_handler; $SIG{__WARN__} = \&warn_handler;
4821     error("File '$pdf_file' seems corrupted. " .
4822         "Perhaps you have to load Memoize earlier in the preamble",
4823         "In particular, Memoize must be loaded before TikZ library " .
4824         "'fadings' and any package deploying it, and in Beamer, " .
4825         "load Memoize by writing \\RequirePackage{memoize} before " .
4826         "\\documentclass{beamer}." .
4827         "This was the error thrown by Perl:" . "\n$0") if $0;
4828 }

```

Does the page exist?

```

4829     if ($ok && $page > (my $n_pages = $pdf->page_count())) {
4830         error("I cannot extract page $page from '$pdf_file', " .
4831             "as it contains only $n_pages page" .
4832             ($n_pages > 1 ? 's' : ''), '');
4833     }
4834     if ($ok) {

```

Import the page into the extern PDF (no disk access yet).

```

4835         my $extern = $pdf_library->new(outver => $pdf->version);
4836         $extern->import_page($pdf, $page);
4837         my $extern_page = $extern->open_page(1);

```

Check whether the page size matches the .mmz expectations.

```

4838         my ($x0, $y0, $x1, $y1) = $extern_page->get_mediabox();
4839         my $width_pt = ($x1 - $x0) / 72 * 72.27;
4840         my $height_pt = ($y1 - $y0) / 72 * 72.27;
4841         my $expected_width_pt = $m_ne{expected_width};
4842         my $expected_height_pt = $m_ne{expected_height};
4843         if ((abs($width_pt - $expected_width_pt) > $tolerance
4844             || abs($height_pt - $expected_height_pt) > $tolerance) && !$force) {
4845             warning("I refuse to extract page $page from $pdf_file, " .
4846                 "because its size (${width_pt}pt x ${height_pt}pt) " .
4847                 "is not what I expected " .
4848                 "(${expected_width_pt}pt x ${expected_height_pt}pt)");
4849         } else {

```

All tests were successful, let's create the extern file. First, the containing directory, if necessary.

```

4850         if ($dir_to_make) {
4851             make_directory($dir_to_make);
4852             $dir_to_make = undef;
4853         }

```

Now the extern file. Note that paranoia\_out was already called above.

```

4854         info("Page $page --> $extern_file_out");
4855         $extern->saveas($extern_file_out);

```

This page will get pruned.

```

4856         $extern_pages{$page} = 1 if $prune;

```

Comment out this \mmzNewExtern.

```

4857         $new_mmz .= '%' unless $keep;

```

```

4858         }
4859     }
4860 }
4861 $new_mmz .= $mmz_line unless $keep;
4862 stderr_to_warning();
4863 }
4864 close(MMZ);
4865 $indent = '';
4866 $texindent = '';
4867 info($done_message);

```

Write out the .mmz file with \mmzNewExtern lines commented out. (All safe, paranoia\_out was already called above.)

```

4868 if (!$keep) {
4869     open(MMZ, ">", $mmz_file);
4870     print MMZ $new_mmz;
4871     close(MMZ);
4872 }

```

Remove the extracted pages from the original PDF. (All safe, paranoia\_out was already called above.)

```

4873 if ($prune and keys(%extern_pages) != 0) {
4874     my $pruned_pdf = $pdf_library->new();
4875     for (my $n = 1; $n <= $pdf->page_count(); $n++) {
4876         if (! $extern_pages{$n}) {
4877             $pruned_pdf->import_page($pdf, $n);
4878         }
4879     }
4880     $pruned_pdf->save($pdf_file);
4881     info("The following extern pages were pruned out of the PDF: " .
4882         join(", ", sort(keys(%extern_pages))));
4883 }
4884
4885 endinput();

```

## 9.2 The Python extraction script memoize-extract.py

```

4886 __version__ = '2024/01/21 v1.1.2'
4887
4888 import argparse, re, sys, os, subprocess, itertools, traceback, platform
4889 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 t the terminal output.

```

4890 ERROR = {
4891     'latex': r'\PackageError{{{package_name}}}{short}{long}',
4892     'plain': r'\errhelp{long}\errmessage{{{package_name}: {short}}}',
4893     'context': r'\errhelp{long}\errmessage{{{package_name}: {short}}}',
4894     None: '{header}{short}.\n{long}',
4895 }
4896
4897 WARNING = {
4898     'latex': r'\PackageWarning{{{package_name}}}{texindent}{text}',
4899     'plain': r'\message{{{package_name}: {texindent}{text}}}',
4900     'context': r'\message{{{package_name}: {texindent}{text}}}',
4901     None: r'{header}{indent}{text}.',
4902 }
4903
4904 INFO = {
4905     'latex': r'\PackageInfo{{{package_name}}}{texindent}{text}',

```

```

4906 'plain': r'\message{{{package_name}: {texindent}{text}}}',
4907 'context': r'\message{{{package_name}: {texindent}{text}}}',
4908 None: r'{header}{indent}{text}.',
4909 }

```

Some variables used in the message routines; note that `header` will be redefined once we parse the arguments.

```

4910 package_name = 'memoize (python-based extraction)'
4911 exit_code = 0
4912 log = None
4913 header = ''
4914 indent = ''
4915 texindent = ''

```

The message routines.

```

4916 def error(short, long):
4917     if not args.quiet:
4918         print(ERROR[None].format(short = short, long = long, header = header))
4919     if log:
4920         long = long.replace('\n', '\\string\\n')
4921         print(
4922             ERROR[args.format].format(
4923                 short = short, long = long, package_name = package_name),
4924             file = log)
4925     global exit_code
4926     exit_code = 11
4927     endinput()
4928
4929 def warning(text):
4930     if log:
4931         print(
4932             WARNING[args.format].format(
4933                 text = text, texindent = texindent, package_name = package_name),
4934             file = log)
4935     if not args.quiet:
4936         print(WARNING[None].format(text = text, header = header, indent = indent))
4937     global exit_code
4938     exit_code = 10
4939
4940 def info(text):
4941     if text and not args.quiet:
4942         print(INFO[None].format(text = text, header = header, indent = indent))
4943     if log:
4944         print(
4945             INFO[args.format].format(
4946                 text = text, texindent = texindent, package_name = package_name),
4947             file = log)

```

Mark the log as complete and exit.

```

4948 def endinput():
4949     if log:
4950         print(r'\endinput', file = log)
4951         log.close()
4952     sys.exit(exit_code)

```

[Permission-related functions](#) `paranoia_in/out` should work exactly as `kpsewhich -safe-in-name/-safe-out-name`.

```

4953 def paranoia_in(f, remark = ''):
4954     if f and not _paranoia(f, openin_any):
4955         error(f"I'm not allowed to read from '{f}' (openin_any = {openin_any})",
4956             remark)
4957
4958 def paranoia_out(f, remark = ''):

```



```

4959     if f and not _paranoia(f, openout_any):
4960         error(f"I'm not allowed to write to '{f}' (openout_any = {openout_any})",
4961             remark)
4962
4963 def _paranoia(f, mode):

    mode is the value of openin_any or openout_any. f is a pathlib.Path object.
4964     return (

    In mode 'any' (a, y or 1), we may access any file.
4965         mode in 'ay1'
4966         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).
4967         not (os.name == 'posix' and f.stem.startswith('.') and f.stem != '.tex')
4968         and (

    If we are precisely in the restricted mode (r, n, 0), then there are no further restrictions.
4969         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.
4970         or (

    We're not allowed to go to a parent directory.
4971         '..' not in f.parts
4972         and

    If the given path is absolute, it should be a descendant of either TEXMF_OUTPUT_DIRECTORY or
    TEXMFOUTPUT.
4973         (not f.is_absolute()
4974         or
4975         is_ancestor(texmf_output_directory, f)
4976         or
4977         is_ancestor(texmfoutput, f)
4978         ))))

    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.
4979 def sanitize_filename(f):
4980     if f and platform.system() == 'Windows' and not (f.is_absolute() or not f.drive):
4981         error(f"\Semi-absolute\" paths are disallowed: '{f}'", r"The path must "
4982             r"either contain both the drive letter and start with '\\', "
4983             r"or none of these; paths like 'C:foo' and '\\foo' are disallowed")
4984
4985 def access_in(f):
4986     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' icaccls 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.
4987 def access_out(f):
4988     try:
4989         exists = f.exists()

    Presumably, we get this error when the parent directory is not executable.
4990     except PermissionError:
4991         return
4992     if exists:

```

An existing file should be writable, and if it's a directory, it should also be executable.

```
4993     return os.access(f, os.W_OK) and (not f.is_dir() or os.access(f, os.X_OK))
4994     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.)

```
4995     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.

```
4996 def find_in(f):
4997     sanitize_filename(f)
4998     if f.is_absolute():
4999         return f
5000     for df in (texmf_output_directory / f if texmf_output_directory else None,
5001              f,
5002              texmfoutput / f if texmfoutput else None):
5003         if df and access_in(df):
5004             return df
5005     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.

```
5006 def find_out(f):
5007     sanitize_filename(f)
5008     if f.is_absolute():
5009         return f
5010     for df in (texmf_output_directory / f if texmf_output_directory else None,
5011              f if not texmf_output_directory else None,
5012              texmfoutput / f if texmfoutput else None):
5013         if df and access_out(df):
5014             return df
5015     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.

```
5016 def is_ancestor(ancestor, descendant):
5017     if not ancestor:
5018         return
5019     a = ancestor.parts
5020     d = descendant.parts
5021     return len(a) < len(d) and a == d[0:len(a)]
```

A paranoid `Path.mkdir`. The given folder is preprocessed by `find_out`.

```
5022 def mkdir(folder):
5023     folder = find_out(Path(folder))
5024     if not folder.exists():
5025         paranoia_out(folder)
```

Using `folder.mkdir` is fine because we know that `TEXMF_OUTPUT_DIRECTORY/TEXMFOUTPUT`, if given, exists, and that "folder" contains no ...

```
5026     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.

```
5027     info(f"Created directory {folder}")
```

```

5028
5029 _re_unquote = re.compile(r'\"(.*)\"')
5030 def unquote(fn):
5031     return _re_unquote.sub(r'\1', fn)

```

**Kpathsea** Get the values of `openin_any`, `openout_any`, `TEXMFOUTPUT` and `TEXMF_OUTPUT_DIRECTORY`.

```

5032 kpsewhich_output = subprocess.run(['kpsewhich',
5033                                     f'-expand-var='
5034                                     f'openin_any=${openin_any},'
5035                                     f'openout_any=${openout_any},'
5036                                     f'TEXMFOUTPUT=${TEXMFOUTPUT}'],
5037                                   capture_output = True
5038                                   ).stdout.decode().strip()
5039 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.

```

5040     openin_any, openout_any = 'p', 'p'
5041     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.

```

5042     warning('I failed to execute "kpsewhich"; , is there no TeX system installed? '
5043             'Assuming openin_any = openout_any = "p" '
5044             '(i.e. restricting all file operations to non-hidden files '
5045             'in the current directory of its subdirectories).')
5046 else:
5047     m = re.fullmatch(r'openin_any=(.*),openout_any=(.*),TEXMFOUTPUT=(.*)',
5048                     kpsewhich_output)
5049     openin_any, openout_any, texmfoutput = m.groups()
5050     texmf_output_directory = os.environ.get('TEXMF_OUTPUT_DIRECTORY', None)
5051     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`.

```

5052     initexmf_output = subprocess.run(
5053         ['initexmf', '--show-config-value=[Core]AllowUnsafeInputFiles',
5054          '--show-config-value=[Core]AllowUnsafeOutputFiles'],
5055         capture_output = True).stdout.decode().strip()
5056     openin_any, openout_any = initexmf_output.split()
5057     openin_any = 'a' if openin_any == 'true' else 'p'
5058     openout_any = 'a' if openout_any == 'true' else 'p'
5059     texmfoutput = None
5060     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`.

```

5061 def sanitize_output_dir(d_str):
5062     d = Path(d_str) if d_str else None
5063     sanitize_filename(d)
5064     return d if d and d.is_dir() and \
5065         (not d.is_absolute() or len(d.parts) != 1 or d.drive) else None
5066
5067 texmfoutput = sanitize_output_dir(texmfoutput)
5068 texmf_output_directory = sanitize_output_dir(texmf_output_directory)
5069
5070 class NotExtracted(UserWarning):
5071     pass

```

We don't delve into the real script when loaded from the testing code.

```
5072 if __name__ == '__main__':
```

## Arguments

```
5073     parser = argparse.ArgumentParser(  
5074         description = "Extract extern pages produced by package Memoize "  
5075             "out of the document PDF.",  
5076         epilog = "For details, see the man page or the Memoize documentation.",  
5077         prog = 'memoize-extract.py',  
5078     )  
5079     parser.add_argument('-P', '--pdf', help = 'extract from file PDF')  
5080     parser.add_argument('-p', '--prune', action = 'store_true',  
5081         help = 'remove the extern pages after extraction')  
5082     parser.add_argument('-k', '--keep', action = 'store_true',  
5083         help = 'do not mark externs as extracted')  
5084     parser.add_argument('-F', '--format', choices = ['latex', 'plain', 'context'],  
5085         help = 'the format of the TeX document invoking extraction')  
5086     parser.add_argument('-f', '--force', action = 'store_true',  
5087         help = 'extract even if the size-check fails')  
5088     parser.add_argument('-q', '--quiet', action = 'store_true',  
5089         help = "describe what's happening")  
5090     parser.add_argument('-m', '--mkdir', action = 'store_true',  
5091         help = 'create a directory (and exit); '  
5092             'mmz argument is interpreted as directory name')  
5093     parser.add_argument('-V', '--version', action = 'version',  
5094         version = f"%(prog)s of Memoize " + __version__)  
5095     parser.add_argument('mmz', help = 'the record file produced by Memoize: '  
5096         'doc.mmz when compiling doc.tex '  
5097         '(doc and doc.tex are accepted as well)')  
5098  
5099     args = parser.parse_args()  
5100  
5101     header = parser.prog + ': ' if args.format else ''
```

Start a new line in the TeX terminal output.

```
5102     if args.format:  
5103         print()
```

**Initialization** With `--mkdir`, argument `mmz` is interpreted as the directory to create.

```
5104     if args.mkdir:  
5105         mkdir(args.mmz)  
5106         sys.exit()
```

Normalize the `mmz` argument into a `.mmz` filename.

```
5107     mmz_file = Path(args.mmz)  
5108     if mmz_file.suffix == '.tex':  
5109         mmz_file = mmz_file.with_suffix('.mmz')  
5110     elif mmz_file.suffix != '.mmz':  
5111         mmz_file = mmz_file.with_name(mmz_file.name + '.mmz')
```

Once we have the `.mmz` filename, we can open the log.

```
5112     if args.format:  
5113         log_file = find_out(mmz_file.with_suffix('.mmz.log'))  
5114         paranoia_out(log_file)  
5115         info(f"Logging to '{log_file}");  
5116         log = open(log_file, 'w')
```

Now that we have opened the log file, we can try loading the PDF processing library.

```
5117     try:  
5118         import pdfwr  
5119     except ModuleNotFoundError:  
5120         error("Python module 'pdfwr' was not found",  
5121             'Have you followed the instructions in section 1.1 of the manual?')
```

Catch any errors in the script and output them to the log.

5122     try:

Find the .mmz file we will read, but retain the original filename in given\_mmz\_file, as we will still need it.

```
5123     given_mmz_file = mmz_file
5124     mmz_file = find_in(mmz_file)
5125     paranoia_in(mmz_file)
5126     if not args.keep:
5127         paranoia_out(mmz_file,
5128             remark = 'This file is rewritten unless option --keep is given.')
```

try:

```
5129     mmz = open(mmz_file)
5130     except FileNotFoundError:
5131         info(f"File '{given_mmz_file}' does not exist, "
5132             f"assuming there's nothing to do")
5133         endinput()
5134     endinput()
```

Determine the PDF filename: it is either given via --pdf, or constructed from the .mmz filename.

```
5135     pdf_file = find_in(Path(args.pdf)
5136                         if args.pdf else given_mmz_file.with_suffix('.pdf'))
5137     paranoia_in(pdf_file)
5138     if args.prune:
5139         paranoia_out(pdf_file,
5140             remark = 'I would have to rewrite this file '
5141             'because option --prune was given.')
```

Various initializations.

```
5142     re_prefix = re.compile(r'\\mmzPrefix *{(?P<prefix>.*?)}')
5143     re_split_prefix = re.compile(r'(?P<dir_prefix>.*/*)?(?P<name_prefix>.*?)')
5144     re_newextern = re.compile(
5145         r'\\mmzNewExtern *{(?P<extern_path>.*?)}{(?P<page_n>[0-9]+)}'
5146         r'{(?P<expected_width>[0-9.]*pt){(?P<expected_height>[0-9.]*pt}')
```

re\_extern\_path = re.compile(

```
5147     r'(?P<dir_prefix>.*/*)?(?P<name_prefix>.*?)'
5148     r'(?P<code_md5sum>[0-9A-F]{32})-'
5149     r'(?P<context_md5sum>[0-9A-F]{32})(?:-[0-9]+)?.pdf')
```

pdf = None

```
5150     extern_pages = []
5151     new_mmz = []
5152     tolerance = 0.01
5153     dir_to_make = None
5154     info(f"Extracting new externs listed in '{mmz_file}' from '{pdf_file}'")
5155     done_message = "Done (there was nothing to extract)"
5156     indent = ' '
5157     texindent = '\space\space '
```

## Process .mmz

```
5160     for line in mmz:
5161         try:
5162             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.

```
5163             prefix = unquote(m_p['prefix'])
5164             if not (m_sp := re_split_prefix.match(prefix)):
5165                 warning(f"Cannot parse line {line.strip()}")
5166             dir_to_make = m_sp['dir_prefix']
5167             elif m_ne := re_newextern.match(line):
```

Found \mmzNewExtern: extract the extern page into an extern file.

```
5168             done_message = "Done"
```

The extern filename, as specified in .mmz:

```

5169         unquoted_extern_path = unquote(m_ne['extern_path'])
5170         extern_file = Path(unquoted_extern_path)

```

We parse the extern filename in a separate step because we have to unquote the entire path.

```

5171         if not (m_ep := re_extern_path.match(unquoted_extern_path)):
5172             warning(f"Cannot parse line {line.strip()}")

```

The actual extern filename:

```

5173         extern_file_out = find_out(extern_file)
5174         paranoia_out(extern_file_out)
5175         page_n = int(m_ne['page_n'])-1

```

Check whether c-memo and cc-memo exist (in any input directory).

```

5176         c_memo = extern_file.with_name(
5177             m_ep['name_prefix'] + m_ep['code_md5sum'] + '.memo')
5178         cc_memo = extern_file.with_name(
5179             m_ep['name_prefix'] + m_ep['code_md5sum']
5180             + '-' + m_ep['context_md5sum'] + '.memo')
5181         c_memo_in = find_in(c_memo)
5182         cc_memo_in = find_in(cc_memo)
5183         if not (access_in(c_memo_in) and access_in(cc_memo_in)) \
5184             and not args.force:
5185             warning(f"I refuse to extract page {page_n+1} into extern "
5186                   f"'{extern_file}', because the associated c-memo "
5187                   f"'{c_memo}' and/or cc-memo '{cc_memo}' "
5188                   f"does not exist")
5189             raise NotExtracted()

```

Load the PDF. We only do this now so that we don't load it if there is nothing to extract.

```

5190         if not pdf:
5191             if not access_in(pdf_file):
5192                 warning(f"Cannot open '{pdf_file}'")
5193             endinput()
5194         try:

```

All safe, paranoia\_in was already called above.

```

5195             pdf = pdfmw.PdfReader(pdf_file)
5196             except pdfmw.errors.PdfParseError as err:
5197                 error(rf"File '{pdf_file}' seems corrupted. Perhaps you "
5198                      rf"have to load Memoize earlier in the preamble",
5199                      f"In particular, Memoize must be loaded before "
5200                      f"TikZ library 'fadings' and any package "
5201                      f"deploying it, and in Beamer, load Memoize "
5202                      f"by writing \RequirePackage{{memoize}} before "
5203                      f"\documentclass{{beamer}}. "
5204                      f"This was the error thrown by Python: \n{err}")

```

Does the page exist?

```

5205         if page_n >= len(pdf.pages):
5206             error(rf"I cannot extract page {page_n} from '{pdf_file}', "
5207                 rf"as it contains only {len(pdf.pages)} page" +
5208                 ('s' if len(pdf.pages) > 1 else ''), '')

```

Check whether the page size matches the .mmz expectations.

```

5209         page = pdf.pages[page_n]
5210         expected_width_pt = float(m_ne['expected_width'])
5211         expected_height_pt = float(m_ne['expected_height'])
5212         mb = page['/MediaBox']
5213         width_bp = float(mb[2]) - float(mb[0])
5214         height_bp = float(mb[3]) - float(mb[1])
5215         width_pt = width_bp / 72 * 72.27
5216         height_pt = height_bp / 72 * 72.27
5217         if (abs(width_pt - expected_width_pt) > tolerance

```

```

5218         or abs(height_pt - expected_height_pt) > tolerance) \
5219         and not args.force:
5220     warning(
5221         f"I refuse to extract page {page_n+1} from '{pdf_file}' "
5222         f"because its size ({width_pt}pt x {height_pt}pt) "
5223         f"is not what I expected "
5224         f"({expected_width_pt}pt x {expected_height_pt}pt)")
5225     raise NotExtracted()

```

All tests were successful, let's create the extern file. First, the containing directory, if necessary.

```

5226         if dir_to_make:
5227             mkdir(dir_to_make)
5228         dir_to_make = None

```

Now the extern file. Note that `paranoia_out` was already called above.

```

5229         info(f"Page {page_n+1} --> {extern_file_out}")
5230         extern = pdfrw.PdfWriter(extern_file_out)
5231         extern.addpage(page)
5232         extern.write()

```

This page will get pruned.

```

5233         if args.prune:
5234             extern_pages.append(page_n)

```

Comment out this `\mmzNewExtern`.

```

5235         if not args.keep:
5236             line = '%' + line
5237     except NotExtracted:
5238         pass
5239     finally:
5240         if not args.keep:
5241             new_mmz.append(line)
5242     mmz.close()
5243     indent = ''
5244     texindent = ''
5245     info(done_message)

```

Write out the `.mmz` file with `\mmzNewExtern` lines commented out. (All safe, `paranoia_out` was already called above.)

```

5246         if not args.keep:
5247             with open(mmz_file, 'w') as mmz:
5248                 for line in new_mmz:
5249                     print(line, file = mmz, end = '')

```

Remove the extracted pages from the original PDF. (All safe, `paranoia_out` was already called above.)

```

5250     if args.prune and extern_pages:
5251         pruned_pdf = pdfrw.PdfWriter(pdf_file)
5252         pruned_pdf.addpages(
5253             page for n, page in enumerate(pdf.pages) if n not in extern_pages)
5254         pruned_pdf.write()
5255         info(f"The following extern pages were pruned out of the PDF: " +
5256             ", ".join(str(page+1) for page in extern_pages))

```

Report that extraction was successful.

```

5257     endinput()

```

Catch any errors in the script and output them to the log.

```

5258     except Exception as err:
5259         error(f'Python error: {err}', traceback.format_exc())

```

### 9.3 The Perl clean-up script `memoize-clean.pl`

```

5260 my $PROG = 'memoize-clean.pl';

```

```

5261 my $VERSION = '2024/01/21 v1.1.2';
5262
5263 use strict;
5264 use Getopt::Long;
5265 use Cwd 'realpath';
5266 use File::Spec;
5267 use File::Basename;
5268
5269 my $usage = "usage: $PROG [-h] [--yes] [--all] [--quiet] [--prefix PREFIX] " .
5270           "[mmz ...]\n";
5271 my $Help = <<END;
5272 Remove (stale) memo and extern files produced by package Memoize.
5273
5274 positional arguments:
5275   mmz                .mmz record files
5276
5277 options:
5278   -h, --help          show this help message and exit
5279   --version, -V       show version and exit
5280   --yes, -y           Do not ask for confirmation.
5281   --all, -a           Remove *all* memos and externs.
5282   --quiet, -q
5283   --prefix PREFIX, -p PREFIX
5284                       A path prefix to clean;
5285                       this option can be specified multiple times.
5286
5287 For details, see the man page or the Memoize documentation.
5288 END
5289
5290 my ($yes, $all, @prefixes, $quiet, $help, $print_version);
5291 GetOptions(
5292   "yes|y"    => \$yes,
5293   "all|a"    => \$all,
5294   "prefix|p=s" => \@prefixes,
5295   "quiet|q|?" => \$quiet,
5296   "help|h|?" => \$help,
5297   "version|V" => \$print_version,
5298   ) or die $usage;
5299 $help and die "$usage\n$Help";
5300 if ($print_version) { print("memoize-clean.pl of Memoize $VERSION\n"); exit 0 }
5301
5302 my (%keep, %prefixes);
5303
5304 my $curdir = Cwd::getcwd();
5305
5306 for my $prefix (@prefixes) {
5307   $prefixes{Cwd::realpath(File::Spec->catfile($curdir, $prefix))} = '';
5308 }
5309
5310 my @mmzs = @ARGV;
5311
5312 for my $mmz (@mmzs) {
5313   my ($mmz_filename, $mmz_dir) = File::Basename::fileparse($mmz);
5314   @ARGV = ($mmz);
5315   my $endinput = 0;
5316   my $empty = -1;
5317   my $prefix = "";
5318   while (<>) {
5319     if (/^ *$/) {
5320     } elsif ($endinput) {
5321       die "Bailing out, \\endinput is not the last line of file $mmz.\n";
5322     } elsif (/^ *\\\mmzPrefix *{(.*)}/) {
5323       $prefix = $1;

```



```

5324     $prefixes{Cwd::realpath(File::Spec->catfile(($curdir,$mmz_dir), $prefix))} = '';
5325     $empty = 1 if $empty == -1;
5326 } elsif (/^%? *\mmz(?:New|Used)(?:CC?Memo|Extern) *{(.*)}/) {
5327     my $fn = $1;
5328     if ($prefix eq '') {
5329 die "Bailing out, no prefix announced before file $fn.\n";
5330     }
5331     $keep{Cwd::realpath(File::Spec->catfile(($mmz_dir), $fn))} = 1;
5332     $empty = 0;
5333     if (rindex($fn, $prefix, 0) != 0) {
5334 die "Bailing out, prefix of file $fn does not match " .
5335     "the last announced prefix ($prefix).\n";
5336     }
5337 } elsif (/^ *\endinput *$/) {
5338     $endinput = 1;
5339 } else {
5340     die "Bailing out, file $mmz contains an unrecognized line: $_\n";
5341 }
5342     }
5343     die "Bailing out, file $mmz is empty.\n" if $empty && !$all;
5344     die "Bailing out, file $mmz does not end with \endinput; this could mean that " .
5345     "the compilation did not finish properly. You can only clean with --all.\n"
5346 if $endinput == 0 && !$all;
5347 }
5348
5349 my @tbdeleted;
5350 sub populate_tbdeleted {
5351     my ($basename_prefix, $dir, $suffix_dummy) = @_;
5352     opendir(MD, $dir) or die "Cannot open directory '$dir'";
5353     while( (my $fn = readdir(MD)) ) {
5354 my $path = File::Spec->catfile(($dir),$fn);
5355 if ($fn =~
5356     /\Q$basename_prefix\E[0-9A-F]{32}(?:-[0-9A-F]{32})?(?:-[0-9]+)?#
5357     (\.memo|(?:-[0-9]+)?\.pdf|\.log)/x
5358     and ($all || !exists($keep{$path}))) {
5359         push @tbdeleted, $path;
5360     }
5361     }
5362     closedir(MD);
5363 }
5364 for my $prefix (keys %prefixes) {
5365     my ($basename_prefix, $dir, $suffix);
5366     if (-d $prefix) {
5367 populate_tbdeleted('', $prefix, '');
5368     }
5369     populate_tbdeleted(File::Basename::fileparse($prefix));
5370 }
5371 @tbdeleted = sort(@tbdeleted);
5372
5373 my @allowed_dirs = ($curdir);
5374 my @deletion_not_allowed;
5375 for my $f (@tbdeleted) {
5376     my $f_allowed = 0;
5377     for my $dir (@allowed_dirs) {
5378 if ($f =~ /\Q$dir\E/) {
5379         $f_allowed = 1;
5380         last;
5381     }
5382     }
5383     push(@deletion_not_allowed, $f) if !$f_allowed;
5384 }
5385 die "Bailing out, I was asked to delete these files outside the current directory:\n" .
5386     join("\n", @deletion_not_allowed) if (@deletion_not_allowed);

```

```

5387
5388 if (scalar(@tbdeleted) != 0) {
5389     my $a;
5390     unless ($yes) {
5391 print("I will delete the following files:\n" .
5392     join("\n",@tbdeleted) . "\n" .
5393     "Proceed (y/n)? ");
5394 $a = lc(<>);
5395 chomp $a;
5396     }
5397     if ($yes || $a eq 'y' || $a eq 'yes') {
5398 foreach my $fn (@tbdeleted) {
5399     print "Deleting ", $fn, "\n" unless $quiet;
5400     unlink $fn;
5401 }
5402     } else {
5403 die "Bailing out.\n";
5404     }
5405 } elsif (!$quiet) {
5406     print "Nothing to do, the directory seems clean.\n";
5407 }

```

## 9.4 The Python clean-up script memoize-clean.py

```

5408 __version__ = '2024/01/21 v1.1.2'
5409
5410 import argparse, re, sys, pathlib, os
5411
5412 parser = argparse.ArgumentParser(
5413     description="Remove (stale) memo and extern files.",
5414     epilog = "For details, see the man page or the Memoize documentation "
5415             "(https://ctan.org/pkg/memoize).")
5416 )
5417 parser.add_argument('--yes', '-y', action = 'store_true',
5418                     help = 'Do not ask for confirmation.')
5419 parser.add_argument('--all', '-a', action = 'store_true',
5420                     help = 'Remove *all* memos and externs.')
5421 parser.add_argument('--quiet', '-q', action = 'store_true')
5422 parser.add_argument('--prefix', '-p', action = 'append', default = [],
5423                     help = 'A path prefix to clean; this option can be specified multiple times.')
5424 parser.add_argument('mmz', nargs= '*', help='mmz record files')
5425 parser.add_argument('--version', '-V', action = 'version',
5426                     version = f"%(prog)s of Memoize " + __version__)
5427 args = parser.parse_args()
5428
5429 re_prefix = re.compile(r'\\mmzPrefix *{(.*)}')
5430 re_memo = re.compile(r'%? *\\mmz(?:New|Used)(?:CC?Memo|Extern) *{(.*)}')
5431 re_endinput = re.compile(r' *\\endinput *$')
5432
5433 prefixes = set(pathlib.Path(prefix).resolve() for prefix in args.prefix)
5434 keep = set()

```

We loop through the given .mmz files, adding prefixes to whatever manually specified by the user, and collecting the files to keep.

```

5435 for mmz_fn in args.mmz:
5436     mmz = pathlib.Path(mmz_fn)
5437     mmz_parent = mmz.parent.resolve()
5438     try:
5439         with open(mmz) as mmz_fh:
5440             prefix = ''
5441             endinput = False
5442             empty = None
5443             for line in mmz_fh:

```

```

5444         line = line.strip()
5445
5446         if not line:
5447             pass
5448
5449         elif endinput:
5450             raise RuntimeError(
5451                 rf'Bailing out, '
5452                 rf'\endinput is not the last line of file {mmz_fn}.')
5453
5454         elif m := re_prefix.match(line):
5455             prefix = m[1]
5456             prefixes.add( (mmz_parent/prefix).resolve() )
5457             if empty is None:
5458                 empty = True
5459
5460         elif m := re_memo.match(line):
5461             if not prefix:
5462                 raise RuntimeError(
5463                     f'Bailing out, no prefix announced before file "{m[1]}".')
5464             if not m[1].startswith(prefix):
5465                 raise RuntimeError(
5466                     f'Bailing out, prefix of file "{m[1]}" does not match '
5467                     f'the last announced prefix ({prefix}).')
5468             keep.add(mmz_parent / m[1])
5469             empty = False
5470
5471         elif re_endinput.match(line):
5472             endinput = True
5473             continue
5474
5475         else:
5476             raise RuntimeError(fr"Bailing out, "
5477                               fr"file {mmz_fn} contains an unrecognized line: {line}")
5478
5479     if empty and not args.all:
5480         raise RuntimeError(fr'Bailing out, file {mmz_fn} is empty.')
5481
5482     if not endinput and empty is not None and not args.all:
5483         raise RuntimeError(
5484             fr'Bailing out, file {mmz_fn} does not end with \endinput; '
5485             fr'this could mean that the compilation did not finish properly. '
5486             fr'You can only clean with --all.'
5487         )

```

It is not an error if the file doesn't exist. Otherwise, cleaning from scripts would be cumbersome.

```

5488     except FileNotFoundError:
5489         pass
5490
5491     tbdeleted = []
5492     def populate_tbdeleted(folder, basename_prefix):
5493         re_aux = re.compile(
5494             re.escape(basename_prefix) +
5495             '[0-9A-F]{32}(?:-[0-9A-F]{32})?'
5496             '(?:-[0-9]+)?(?:\.memo|(?:-[0-9]+)?\.pdf|\\.log)$')
5497         try:
5498             for f in folder.iterdir():
5499                 if re_aux.match(f.name) and (args.all or f not in keep):
5500                     tbdeleted.append(f)
5501         except FileNotFoundError:
5502             pass
5503
5504     for prefix in prefixes:

```

```

    "prefix" is interpreted both as a directory (if it exists) and a basename prefix.
5505     if prefix.is_dir():
5506         populate_tbdeleted(prefix, '')
5507     populate_tbdeleted(prefix.parent, prefix.name)
5508
5509 allowed_dirs = [pathlib.Path().absolute()] # todo: output directory
5510 deletion_not_allowed = [f for f in tbdeleted if not f.is_relative_to(*allowed_dirs)]
5511 if deletion_not_allowed:
5512     raise RuntimeError("Bailing out, "
5513         "I was asked to delete these files outside the current directory:\n" +
5514         "\n".join(str(f) for f in deletion_not_allowed))
5515
5516 _cwd_absolute = pathlib.Path().absolute()
5517 def relativize(path):
5518     try:
5519         return path.relative_to(_cwd_absolute)
5520     except ValueError:
5521         return path
5522
5523 if tbdeleted:
5524     tbdeleted.sort()
5525     if not args.yes:
5526         print('I will delete the following files:')
5527         for f in tbdeleted:
5528             print(relativize(f))
5529         print("Proceed (y/n)? ")
5530         a = input()
5531     if args.yes or a == 'y' or a == 'yes':
5532         for f in tbdeleted:
5533             if not args.quiet:
5534                 print("Deleting", relativize(f))
5535             try:
5536                 f.unlink()
5537             except FileNotFoundError:
5538                 print(f"Cannot delete {f}")
5539     else:
5540         print("Bailing out.")
5541 elif not args.quiet:
5542     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)	2041	inner handler	2100
/handlers/.meaning to context (/mmz)	829	integrated driver	696
/handlers/.value to context (/mmz)	829	memoize	1679
/advice/install keys:		multiref	1767
activation	2053	noop	1687
setup key	2053	options	2100
/advice keys:		outer handler	2100
activation/deferred	2202	raw collector options	2100
activation/immediate	2202	ref	1741
/collargs keys:		refrange	1754
append expandable postprocessor	2779	replicate	1701
append expandable preprocessor	2779	reset	2124
append postprocessor	2750	run conditions	2100
append postwrap	2805	run if memoization is possible	1650
append preprocessor	2750	run if memoizing	1658
append prewrap	2805	to context	1716
begin tag	2734	/mmz keys:	
brace collected	2828	/handlers/.meaning to context	829
braces	2726	/handlers/.value to context	829
caller	2660	activate	2213
clear postprocessors	2771	activate csname	2088
clear preprocessors	2771	activate deferred	2087
end tag	2734	activate key	2090
environment	2729	activation	2085
fix from no verbatim	2707	after memoization	643
fix from verb	2707	at begin memoization	643
fix from verbatim	2707	at end memoization	643
ignore nesting	2742	auto	2062
ignore other tags	2746	auto csname	2062
no delimiters	2824	auto csname'	2062
no verbatim	2697	auto key	2062
prepend expandable postprocessor	2779	auto key'	2062
prepend expandable preprocessor	2779	auto'	2062
prepend postprocessor	2750	bat	1371
prepend postwrap	2805	begindocument	1945
prepend preprocessor	2750	begindocument/before	1945
prepend prewrap	2805	begindocument/end	1945
tags	2734	capture	684
verb	2697	clear context	811
verbatim	2697	context	811
verbatim ranges	3834	csname meaning to context	829
/handlers keys:		deactivate	2213
.install advice	2041	deactivate csname	2088
/mmz/auto keys:		deactivate key	2090
after setup	2125	direct ccmemo input	929
apply options	1666	disable	240
args	2100	driver	672
bailout handler	2100	enable	240
clear collector options	2100	enddocument/afterlastpage	1945
clear options	2100	extract	1239
clear raw collector options	2100	extract/perl	1244
collector	2100	extract/python	1244
collector options	2100	extract/tex	1427
force multiref	1767	force activate	2213
force ref	1741	ignore spaces	341
force refrange	1754	include context in ccmemo	925
		include source in cmemo	872

key meaning to context	829	\advice@init@I	2178, 2179, 2181, 2337, 2438, 2482, 2486
key value to context	829	\advice@init@i	2178, 2178, 2180, 2317, 2422, 2481, 2485
makefile	1399	\advice@original@cs	2230, 2231, 2327, 2417, 2428, 2498
manual	1602	\advice@original@csname	2230, 2230, 2266,
meaning to context	829		2285, 2307, 2312, 2313, 2398, 2401, 2406, 2407, 2414
memo dir	306	\advice@pgfkeys@collector	2078, 2083, 2377, 2377
mkdir	289	\advice@setup@save	2153, 2182, 2182
mkdir command	289	\advice@trace	2488, 2494, 2496, 2497, 2502,
no memo dir	306		2510, 2519, 2525, 2528, 2531, 2534, 2540, 2543, 2546
no record	1317	\advice@trace@init@I	2482, 2495, 2516
no verbatim	345	\advice@trace@init@i	2481, 2495, 2495
normal	254	\advice@timeout	717, 2488, 2489, 2491, 2492, 2494
options	252	\AdviceArgs	
padding	1026		2108, 2167, 2175, 2195, 2336, 2355, 2365, 2371, 2530
padding bottom	1021	\AdviceBailoutHandler	
padding left	1021		2101, 2157, 2187, 2315, 2326, 2427, 2509, 2512, 2513
padding right	1021	\AdviceCollector	
padding to context	1031		1669, 2103, 2159, 2191, 2336, 2350, 2524, 2527, 2537
padding top	1021	\AdviceCollectorOptions	
prefix	263		2104, 2105, 2160, 2194, 2336, 2361, 2362, 2533
readonly	254	\AdviceCsnameGetOriginal	1722, 2239
recompile	254	\AdviceGetOriginal	2230, 2336, 2437
record	1301	\AdviceIfArgs	2353, 2369
record/bat/...	1387	\AdviceInnerHandler	2109,
record/makefile/...	1406		2161, 2196, 2336, 2366, 2378, 2542, 2549, 2551, 2583
record/mMZ/...	1339	\AdviceName	1694, 2129,
record/sh/...	1375		2172, 2330, 2333, 2335, 2346, 2354, 2359, 2434, 2478
sh	1371	\AdviceNamespace	1719, 1722, 2330, 2332, 2433, 2477
tex extraction command	1462	\AdviceOptions	
tex extraction options	1462		1675, 1704, 2110, 2111, 2113, 2162, 2197, 2336, 2548
tex extraction script	1462	\AdviceOriginal	
tracing	762		1683,
try activate	2213		1688, 1698, 1710, 1727, 1744, 1765, 1777, 1783,
verb	345		1796, 1894, 1901, 1903, 1940, 1942, 2330, 2336, 2437
verbatim	345	\AdviceOuterHandler	
			2102, 2158, 2190, 2336, 2338, 2439, 2518, 2521, 2522
		\AdviceRawCollectorOptions	
			1677,
			2106, 2107, 2166, 2171, 2192, 2193, 2336, 2360, 2536
		\AdviceReplaced	1682, 1709, 2330, 2335, 2436
		\AdviceRunConditions	
			2100, 2156, 2186, 2315, 2319, 2423, 2500, 2501, 2504
		\AdviceSetup	2063, 2067, 2129
		\AdviceTracingOff	2480
		\AdviceTracingOn	2480
		\AdviceType	1626, 2124, 2129
		after memoization (/mmz)	643
		after setup (/mmz/auto)	2125
		append expandable postprocessor (/collargs)	2779
		append expandable preprocessor (/collargs)	2779
		append postprocessor (/collargs)	2750
		append postwrap (/collargs)	2805
		append preprocessor (/collargs)	2750
		append prewrap (/collargs)	2805
		apply options (/mmz/auto)	1666
		args (/mmz/auto)	2100
		at begin memoization (/mmz)	643
		at end memoization (/mmz)	643
		auto (/mmz)	2062
		auto csname (/mmz)	2062
		auto csname' (/mmz)	2062
		auto key (/mmz)	2062
		auto key' (/mmz)	2062
		auto' (/mmz)	2062
A			
abortOnError (Lua function)	564		
activate (/mmz)	2213		
activate csname (/mmz)	2088		
activate deferred (/mmz)	2087		
activate key (/mmz)	2090		
activation (/advice/install)	2053		
activation (/mmz)	2085		
activation/deferred (/advice)	2202		
activation/immediate (/advice)	2202		
\advice@activate	2215, 2252, 2252		
\advice@activate@cmd	2254, 2264, 2264, 2383		
\advice@activate@cmd@do	2271, 2277, 2306, 2306		
\advice@activate@env	2256, 2380, 2381, 2397		
\advice@begin@env@outer	2425, 2432, 2432		
\advice@begin@env@rc	2415, 2421, 2421		
\advice@begin@rc	2126, 2413, 2413		
\advice@CollectArgumentsRaw	2118, 2159, 2352, 2352		
\advice@deactivate	2220, 2252, 2258		
\advice@deactivate@cmd	2260, 2283, 2283, 2390		
\advice@deactivate@cmd@do	2287, 2311, 2311		
\advice@deactivate@env	2262, 2380, 2388, 2405		
\advice@default@outer@handler	2117, 2158, 2349, 2349		
\advice@handle	2299, 2308, 2316, 2316		
\advice@handle@outer	2324, 2331, 2331		
\advice@handle@rc	2320, 2322, 2322, 2413		
\advice@if@our@definition	2267, 2286, 2298, 2298		

<b>B</b>	
bailout handler (/mmz/auto) .....	2100
bat (/mmz) .....	1371
begin tag (/collargs) .....	2734
begindocument (/mmz) .....	1945
begindocument/before (/mmz) .....	1945
begindocument/end (/mmz) .....	1945
brace collected (/collargs) .....	2828
braces (/collargs) .....	2726
<b>C</b>	
caller (/collargs) .....	2660
capture (/mmz) .....	684
\catcodetable@atletter .....	3799, 3802
clear collector options (/mmz/auto) .....	2100
clear context (/mmz) .....	811
clear options (/mmz/auto) .....	2100
clear postprocessors (/collargs) .....	2771
clear preprocessors (/collargs) .....	2771
clear raw collector options (/mmz/auto) .....	2100
\collargs@ .....	2640, 2833, 3103, 3111, 3133, 3153, 3174, 3180, 3225, 3232, 3236, 3276, 3282, 3334, 3354, 3369, 3375, 3398, 3406, 3522, 3710, 3723
\collargs@& .....	2867
\collargs@! .....	2892
\collargs@+ .....	2886
\collargs@. .....	2898
\collargs@@ .....	2835, 2837, 2880, 2884, 2891
\collargs@appendarg .....	2913, 2913, 3102, 3109, 3132, 3152, 3179, 3218, 3222, 3235, 3275, 3281, 3333, 3352, 3368, 3396, 3404, 3521, 3708, 3709
\collargs@b .....	3411, 3411
\collargs@bgroups ..	3752, 3761, 3773, 3773, 3869, 3912
\collargs@catcodetable@initex ..	3799, 3811, 3813, 3819
\collargs@catcodetable@verbatim .....	3799, 3801, 3806, 3807, 3810, 3812, 3830, 3865
\collargs@cc 2994, 2997, 3317, 3325, 3342, 3775, 3776, 3781	
\collargs@cs@cases 840, 2133, 2253, 2259, 2643, 2643, 2664	
\collargs@D .....	3193, 3193
\collargs@d .....	3160, 3160, 3193, 3194, 3195
\collargs@defcollector ..	2930, 2930, 3149, 3176, 3402
\collargs@defusecollector .....	2930, 2934, 3101, 3107, 3130, 3150, 3177, 3273, 3280
\collargs@E .....	3724, 3724
\collargs@e .....	3659, 3659, 3724
\collargs@egroups ..	3753, 3763, 3773, 3774, 3871, 3923
\collargs@fix .....	2847, 2902, 3949, 3949, 4233
\collargs@fix@NtoN .....	3970, 3970-3972
\collargs@fix@NtoV .....	3983, 3988, 3988, 4037, 4040
\collargs@fix@Ntov .....	3973, 3973
\collargs@fix@VtoN .....	4028, 4048, 4059, 4059
\collargs@fix@vtoN .....	4018, 4018
\collargs@fix@VtoN@comment .....	4064, 4074, 4093
\collargs@fix@VtoN@escape .....	4061, 4097, 4097
\collargs@fix@VtoN@token .....	4066, 4071, 4071
\collargs@fix@VtoV .....	3970, 3972
\collargs@fix@Vtov .....	4046
\collargs@fix@vtoV .....	4032, 4032
\collargs@fix@vtov .....	3970, 3971
\collargs@forrange .....	3066, 3066, 3084, 3828, 3829
\collargs@forranges .....	3082, 3082, 3884, 3940
\collargs@G .....	3380, 3380
\collargs@g .....	3371, 3371, 3380
\collargs@grabbed@spaces .....	2924, 3030, 3030, 3043, 3222, 3594, 3620
\collargs@grabspaces .....	2950, 2950, 3025, 3157, 3216, 3259, 3286, 3385, 3504, 3584, 3615, 3673
\collargs@ifnextcat .....	3046, 3046, 3412
\collargs@init@grabspaces .....	2637, 2925, 2945, 2945, 3040, 3221, 3595, 3621
\collargs@insert@char 4072, 4249, 4250, 4261, 4271, 4296	
\collargs@l .....	3089, 3089
\collargs@letusecollector .....	2930, 2939, 3557, 3585, 3616, 3627, 3640, 3655
\collargs@m .....	3251, 3251, 3378, 3711
\collargs@make@char .....	3916, 3927, 4249, 4255, 4265, 4275, 4305
\collargs@make@no@verbatim ...	3742, 3933, 3934, 3938
\collargs@make@verbatim ..	3729, 3735, 3839, 3840, 3875
\collargs@make@verbatim@bgroup .....	3892, 3908, 3908
\collargs@make@verbatim@comment .....	3898, 3930, 3930
\collargs@make@verbatim@egroup .....	3895, 3919, 3919
\collargs@maybegrabspaces .....	3023, 3023, 3191, 3245
\collargs@O .....	3195, 3195
\collargs@o .....	3194, 3194
\collargs@other@bgroup .....	3108, 3112, 3300, 3358, 3366, 3388, 3389, 3394, 3448, 3459, 3789, 3794, 3916
\collargs@other@egroup .....	3303, 3359, 3366, 3390, 3394, 3448, 3459, 3789, 3795, 3927
\collargs@percentchar 881, 883, 939, 951, 3085, 3087, 3844	
\collargs@R .....	3159, 3159
\collargs@r .....	3138, 3138, 3159
\collargs@readContent ...	3362, 3400, 3436, 3543, 3543
\collargs@reinsert@spaces 3037, 3037, 3090, 3127, 3436	
\collargs@s .....	3250, 3250
\collargs@t .....	3196, 3196, 3250
\collargs@toks .....	2639, 2896, 2896, 2905, 2907, 2924
\collargs@u .....	3116, 3116
\collargs@v .....	3381, 3381
\collargs@verbatim@ranges .....	3834, 3835, 3837, 3838, 3843, 3884, 3940
\collargs@verbatim@wrap ..	2636, 2697, 2702, 2879, 2883
\collargs@wrap .....	2928, 2928, 3131, 3151, 3178, 3274, 3367, 3395, 3403, 3541, 3714
\collargsAppendExpandablePostprocessor .....	2779
\collargsAppendExpandablePreprocessor .....	2779
\collargsAppendPostprocessor .....	2750
\collargsAppendPostwrap .....	326, 440, 2805
\collargsAppendPreprocessor .....	2750
\collargsAppendPrewrap .....	2805
\collargsArg .....	2767, 2800, 2801, 2818, 2820, 2897, 2914, 2924, 3337, 3343, 3352, 3368, 3396, 3521, 3555, 3600, 3632, 3649, 3709, 3719, 4242
\collargsBraces .....	2727, 3383, 3789
\collargsCaller .....	389, 436, 2359, 2660
\collargsClearPostprocessors .....	2771
\collargsClearPreprocessors .....	2771
\collargsEnvironment .....	2172, 2729
\collargsFixFromNoVerbatim .....	368, 401, 421, 2707
\collargsFixFromVerb .....	2707
\collargsFixFromVerbatim .....	2707
\collargsNoVerbatim .....	357, 2700, 3725
\collargsPrependExpandablePostprocessor .....	2779
\collargsPrependExpandablePreprocessor .....	2779
\collargsPrependPostprocessor .....	2750
\collargsPrependPostwrap .....	2805

<code>\collargsPrependPreprocessor</code> .....	2750	<code>\ifcollargsIgnoreOtherTags</code> .....	2746, 3426, 3496
<code>\collargsPrependPrewrap</code> .....	2805	<code>\ifcollargsNoDelimiters</code> .....	2824, 2919
<code>\collargsSet</code> .....	2362,	<code>\ifinmemoize</code> .....	491, 561, 1652
2630, 2642, 2660, 2697, 2707, 2726, 2729, 2734, 2742,		<code>\ifmemoize</code> .....	240, 525, 723, 1651
2746, 2750, 2773, 2779, 2805, 2824, 2828, 2883, 3834		<code>\IfMemoizing</code> .....	705, 2004
<code>\collargsVerb</code> .....	353, 2699, 3725	<code>\ifmemoizing</code> .....	489, 523, 648, 656, 664, 813, 821, 1659, 1840
<code>\collargsVerbatim</code> .....	349, 2698, 3384, 3725	<code>\ifmmz@abort</code> .....	537, 619, 640, 640, 1002
<code>\collargsVerbatimRanges</code> .....	3834	<code>\ifmmz@direct@ccmemo@input</code> .....	
<code>\CollectArguments</code> .....	2627, 2694	.....	753, 929, 929, 955, 972, 982, 992
<code>\CollectArgumentsRaw</code> .....	323, 388, 435, 2358, 2627	<code>\ifmmz@include@context</code> .....	925, 925, 940
<code>collector (/mmz/auto)</code> .....	2100	<code>\ifmmz@include@source</code> .....	872, 875, 884
<code>collector options (/mmz/auto)</code> .....	2100	<code>\ifmmzkeepexterns</code> .....	676, 1073
<code>context (/mmz)</code> .....	811	<code>\ifmmzUnmemoizable</code> .....	534, 612, 642
<code>csname meaning to context (/mmz)</code> .....	829	<code>ignore nesting (/collargs)</code> .....	2742
		<code>ignore other tags (/collargs)</code> .....	2746
		<code>ignore spaces (/mmz)</code> .....	341
		<code>include context in ccmemo (/mmz)</code> .....	925
		<code>include source in cmemo (/mmz)</code> .....	872
		<code>inner handler (/mmz/auto)</code> .....	2100
		<code>integrated driver (/mmz/auto)</code> .....	696
		<b>K</b>	
		<code>key meaning to context (/mmz)</code> .....	829
		<code>key value to context (/mmz)</code> .....	829
		<b>L</b>	
		<code>Lua functions:</code>	
		<code>abortOnError</code> .....	564
		<b>M</b>	
		<code>makefile (/mmz)</code> .....	1399
		<code>manual (/mmz)</code> .....	1602
		<code>meaning to context (/mmz)</code> .....	829
		<code>memo dir (/mmz)</code> .....	306
		<code>\Memoize</code> .....	384, 432, 508, 722, 1681
		<code>\memoize</code> .....	410, 483
		<code>memoize (/mmz/auto)</code> .....	1679
		<code>memoize (env.)</code> .....	395
		<code>\memoizinggrouplevel</code> .....	595, 638, 707
		<code>mkdir (/mmz)</code> .....	289
		<code>mkdir command (/mmz)</code> .....	289
		<code>\mmz</code> .....	361, 452
		<code>\mmz@auto@bailout</code> .....	1617, 1629, 1666, 1671
		<code>\mmz@auto@label</code> .....	1738, 1779, 1779
		<code>\mmz@auto@memoize</code> .....	1623, 1679, 1679
		<code>\mmz@auto@multiref</code> .....	1767, 1769, 1771, 1774
		<code>\mmz@auto@noop</code> .....	1627, 1631, 1687, 1687
		<code>\mmz@auto@noop@env</code> .....	1629, 1687, 1693
		<code>\mmz@auto@outer</code> .....	1618, 1666, 1666
		<code>\mmz@auto@rc@if@memoization@possible</code> .....	1613, 1650, 1650
		<code>\mmz@auto@rc@if@memoizing</code> .....	1615, 1658, 1658
		<code>\mmz@auto@ref</code> .....	1732, 1733, 1741, 1741
		<code>\mmz@auto@refrange</code> .....	1754, 1756, 1758, 1761
		<code>\mmz@auto@replicate</code> .....	1633, 1701, 1701
		<code>\mmz@auto@tocontext</code> .....	1634, 1716, 1716
		<code>\mmz@biblatex@cite</code> .....	1885, 1896, 1896
		<code>\mmz@biblatex@cites</code> .....	1886, 1909, 1909
		<code>\mmz@biblatex@entry</code> .....	1884, 1891, 1891
		<code>\mmz@ccmemo@append@resource</code> .....	948, 948, 1058
		<code>\mmz@ccmemo@resources</code> .....	600, 936, 1057, 1064, 1064
		<code>\mmz@compile</code> .....	522, 555, 555, 1005
		<code>\mmz@compute@context@mdfivesum</code> .....	622, 625, 904, 906, 906
		<code>\mmz@if@roughly@equal</code> .....	1185, 1186, 1528, 1531, 1558, 1559
<code>\collargsPrependPreprocessor</code> .....	2750		
<code>\collargsPrependPrewrap</code> .....	2805		
<code>\collargsSet</code> .....	2362,		
2630, 2642, 2660, 2697, 2707, 2726, 2729, 2734, 2742,			
2746, 2750, 2773, 2779, 2805, 2824, 2828, 2883, 3834			
<code>\collargsVerb</code> .....	353, 2699, 3725		
<code>\collargsVerbatim</code> .....	349, 2698, 3384, 3725		
<code>\collargsVerbatimRanges</code> .....	3834		
<code>\CollectArguments</code> .....	2627, 2694		
<code>\CollectArgumentsRaw</code> .....	323, 388, 435, 2358, 2627		
<code>collector (/mmz/auto)</code> .....	2100		
<code>collector options (/mmz/auto)</code> .....	2100		
<code>context (/mmz)</code> .....	811		
<code>csname meaning to context (/mmz)</code> .....	829		
<b>D</b>			
<code>deactivate (/mmz)</code> .....	2213		
<code>deactivate csname (/mmz)</code> .....	2088		
<code>deactivate key (/mmz)</code> .....	2090		
<code>direct ccmemo input (/mmz)</code> .....	929		
<code>disable (/mmz)</code> .....	240		
<code>driver (/mmz)</code> .....	672		
<b>E</b>			
<code>enable (/mmz)</code> .....	240		
<code>end tag (/collargs)</code> .....	2734		
<code>enddocument/afterlastpage (/mmz)</code> .....	1945		
<code>environment (/collargs)</code> .....	2729		
<code>environments:</code>			
<code>memoize</code> .....	395		
<code>nomemoize</code> .....	453		
<code>\etoksapp</code> .....	2608, 3572, 3594, 3600, 3620, 3649		
<code>extract (/mmz)</code> .....	1239		
<code>extract/perl (/mmz)</code> .....	1244		
<code>extract/python (/mmz)</code> .....	1244		
<code>extract/tex (/mmz)</code> .....	1427		
<b>F</b>			
<code>\filetotoks</code> .....	214, 995		
<code>fix from no verbatim (/collargs)</code> .....	2707		
<code>fix from verb (/collargs)</code> .....	2707		
<code>fix from verbatim (/collargs)</code> .....	2707		
<code>force activate (/mmz)</code> .....	2213		
<code>force multiref (/mmz/auto)</code> .....	1767		
<code>force ref (/mmz/auto)</code> .....	1741		
<code>force refrange (/mmz/auto)</code> .....	1754		
<b>G</b>			
<code>\gtoksapp</code> .....	614, 814, 1708, 1850, 1868, 2608		
<b>I</b>			
<code>\ifAdviceRun</code> .....	2323, 2348, 2424, 2506		
<code>\ifcollargs@fix@requested</code> .....	3948, 3948, 3951		
<code>\ifcollargs@verbatim</code> .....	2633,		
2695, 2695, 2851, 2974, 3031, 3117, 3139, 3161,			
3197, 3252, 3419, 3441, 3452, 3467, 3478, 3662, 3959			
<code>\ifcollargs@verbatimbraces</code> .....			
.....	2634, 2695, 2696, 2852, 3094, 3289, 3424,		
3447, 3458, 3493, 3530, 3537, 3866, 3909, 3920, 3960			
<code>\ifcollargsAddTags</code> .....	2734		
<code>\ifcollargsBeginTag</code> .....	2734, 3527		
<code>\ifcollargsBraceCollected</code> .....	2828, 2904		
<code>\ifcollargsEndTag</code> .....	2734, 3534		
<code>\ifcollargsIgnoreNesting</code> .....	2742, 3631		



<code>\mmz@include@extern</code>	969, 1167, 1167	<code>multiref (/mmz/auto)</code>	1767
<code>\mmz@include@extern@from@tbe@box</code>	634, 1222, 1222		
<code>\mmz@insertpdfpage</code>	1174, 1203, 1204, 1211, 1215		
<code>\mmz@inverse@mag</code>			
	1134, 1139, 1141, 1143, 1145–1148, 1150, 1152		
<code>\mmz@lua@atbeginmemoization</code>	564, 576, 588, 606		
<code>\mmz@lua@atendmemoization</code>	564, 579, 589, 611		
<code>\mmz@maybe@scantokens</code>	492, 494, 519, 1688		
<code>\mmz@memoize</code>	530, 542, 591, 591, 1007		
<code>\mmz@openin</code>	215, 224, 224, 1876		
<code>\mmz@openout</code>	224,		
	225, 878, 934, 1285, 1342, 1377, 1389, 1410, 1877		
<code>\mmz@process@ccmemo</code>	547, 963, 963		
<code>\mmz@process@cmemo</code>	533, 892, 892		
<code>\mmz@remove@quotes</code>	300, 313, 313		
<code>\mmz@shipout@extern</code>	1076, 1080, 1080		
<code>\mmz@shipout@externs</code>	628, 1066, 1066		
<code>\mmz@shipout@mag</code>	1088, 1132, 1132		
<code>\mmz@split@prefix</code>	266		
<code>\mmz@tbe@box</code>	599, 1056, 1065, 1065, 1073, 1226		
<code>\mmz@write@ccmemo</code>	627, 933, 933		
<code>\mmz@write@cmemo</code>	615, 623, 626, 877, 877		
<code>\mmzAbort</code>	639, 1637, 1751, 1907, 2002		
<code>\mmzAfterMemoization</code>	635, 643		
<code>\mmzAtBeginMemoization</code>	607, 643, 1867		
<code>\mmzAtEndMemoization</code>	609, 643		
<code>\mmzAutoInit</code>	1668, 1674, 1695		
<code>\mmzCCMemo</code>	602, 678, 681, 924,		
	939, 1708, 1780, 1811, 1850, 1854, 1899, 1938, 2010		
<code>\mmzCMemo</code>	601, 614, 871, 883, 1848, 1868, 2009		
<code>\mmzContext</code>	809, 816, 824, 907, 2007		
<code>\mmzContextExtra</code>			
	603, 613, 809, 814, 822, 881, 897, 907, 2008		
<code>\mmzEndMemo</code>	944, 958, 963		
<code>\mmzExternalizeBox</code>	680, 1039		
<code>\mmzExternPages</code>	1122, 1127, 1157, 1160–1162, 2011		
<code>\mmzExtraPages</code>	1127, 1166, 1793, 1794, 1796, 2012		
<code>\mmzForceNoRef</code>	1733, 1746, 1758, 1771, 2006		
<code>\mmzLabel</code>	1781, 1785		
<code>\mmzNewCCMemo</code>	1356, 1434		
<code>\mmzNewCMemo</code>	1353, 1434		
<code>\mmzNewExtern</code>	1349, 1440		
<code>\mmznxt</code>	1661, 1672, 1676		
<code>\mmzNoRef</code>	1732, 1746, 1756, 1769, 2005		
<code>\mmzPrefix</code>	1345, 1434		
<code>\mmzRegularPages</code>	129, 1125, 1154		
<code>\mmzResource</code>	951, 953		
<code>\mmzset</code>	230, 244,		
	258, 263, 283, 289, 306, 342, 347, 373, 404, 426, 646,		
	672, 684, 696, 762, 811, 872, 926, 930, 1021, 1129,		
	1151, 1158, 1240, 1301, 1317, 1331, 1333, 1339, 1406,		
	1427, 1467, 1598, 1603, 1636, 1639, 1643, 1645, 1675,		
	1676, 1730, 1735, 1754, 1767, 1795, 1798, 1805, 1824,		
	1827, 1837, 1882, 1946–1949, 1952, 1957, 1967, 1974		
<code>\mmzSingleExternDriver</code>	674, 677		
<code>\mmzSource</code>	885, 891		
<code>\mmzThisContext</code>	941, 963		
<code>\mmzTracingOff</code>	762, 2014		
<code>\mmzTracingOn</code>	762, 2013		
<code>\mmzUnmemoizable</code>	641, 1640, 2003		
<code>\mmzUsedCCMemo</code>	1365, 1434		
<code>\mmzUsedCMemo</code>	1362, 1434		
<code>\mmzUsedExtern</code>	1359, 1434		
		<b>N</b>	
		<code>no delimiters (/collargs)</code>	2824
		<code>no memo dir (/mmz)</code>	306
		<code>no record (/mmz)</code>	1317
		<code>no verbatim (/collargs)</code>	2697
		<code>no verbatim (/mmz)</code>	345
		<code>\nomemoize</code>	462, 483
		<code>nomemoize (env.)</code>	453
		<code>\nommz</code>	443
		<code>\nommzkeys</code>	235
		<code>noop (/mmz/auto)</code>	1687
		<code>normal (/mmz)</code>	254
		<b>O</b>	
		<code>options (/mmz/auto)</code>	2100
		<code>options (/mmz)</code>	252
		<code>outer handler (/mmz/auto)</code>	2100
		<b>P</b>	
		<code>padding (/mmz)</code>	1026
		<code>padding bottom (/mmz)</code>	1021
		<code>padding left (/mmz)</code>	1021
		<code>padding right (/mmz)</code>	1021
		<code>padding to context (/mmz)</code>	1031
		<code>padding top (/mmz)</code>	1021
		<code>prefix (/mmz)</code>	263
		<code>prepend expandable postprocessor (/collargs)</code>	2779
		<code>prepend expandable preprocessor (/collargs)</code>	2779
		<code>prepend postprocessor (/collargs)</code>	2750
		<code>prepend postwrap (/collargs)</code>	2805
		<code>prepend preprocessor (/collargs)</code>	2750
		<code>prepend prewrap (/collargs)</code>	2805
		<b>R</b>	
		<code>raw collector options (/mmz/auto)</code>	2100
		<code>readonly (/mmz)</code>	254
		<code>recompile (/mmz)</code>	254
		<code>record (/mmz)</code>	1301
		<code>record/bat/... (/mmz)</code>	1387
		<code>record/makefile/... (/mmz)</code>	1406
		<code>record/mmz/... (/mmz)</code>	1339
		<code>record/sh/... (/mmz)</code>	1375
		<code>\ref</code>	1736
		<code>ref (/mmz/auto)</code>	1741
		<code>refrange (/mmz/auto)</code>	1754
		<code>replicate (/mmz/auto)</code>	1701
		<code>reset (/mmz/auto)</code>	2124
		<code>run conditions (/mmz/auto)</code>	2100
		<code>run if memoization is possible (/mmz/auto)</code>	1650
		<code>run if memoizing (/mmz/auto)</code>	1658
		<b>S</b>	
		<code>setup key (/advice/install)</code>	2053
		<code>sh (/mmz)</code>	1371
		<b>T</b>	
		<code>tags (/collargs)</code>	2734
		<code>tex extraction command (/mmz)</code>	1462
		<code>tex extraction options (/mmz)</code>	1462
		<code>tex extraction script (/mmz)</code>	1462
		<code>to context (/mmz/auto)</code>	1716

\toksapp .....	220, 816,	verbatim (/collargs) .....	2697
	1934, 2570, 2574, 2580, 2581, 2608, 3343, 3551, 3626	verbatim (/mmz) .....	345
tracing (/mmz) .....	762	verbatim ranges (/collargs) .....	3834
try activate (/mmz) .....	2213		

**V**

verb (/collargs) .....	2697
verb (/mmz) .....	345

**X**

\xtoksapp .....	678, 681, 1057,
	1420, 1780, 1811, 1848, 1854, 1899, 1938, 2608, 2924