

# The zref-clever package

## User manual

gusbrs

<https://github.com/gusbrs/zref-clever>  
<https://www.ctan.org/pkg/zref-clever>

Version v0.4.8 – 2024-11-07

### **Abstract**

zref-clever provides a user interface for making  $\LaTeX$  cross-references which automates some of their typical features, thus easing their input in the document and improving the consistency of typeset results. A reference made with `\zcref` includes a “name” according to its “type” and lists of multiple labels can be automatically sorted and compressed into ranges when due. The reference format is highly and easily customizable, both globally and locally. zref-clever is based on zref’s extensible referencing system.

### **EXPERIMENTAL**

Please read Section 2 carefully.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Warning</b>	<b>4</b>
<b>3</b>	<b>zref-clever for the impatient</b>	<b>4</b>
<b>4</b>	<b>Loading the package</b>	<b>5</b>
	4.1 Dependencies . . . . .	5
<b>5</b>	<b>User interface</b>	<b>5</b>
<b>6</b>	<b>Options</b>	<b>6</b>
<b>7</b>	<b>Reference types</b>	<b>12</b>
<b>8</b>	<b>Reference format</b>	<b>13</b>
	8.1 Advanced reference formatting . . . . .	17
<b>9</b>	<b>Internationalization</b>	<b>18</b>
<b>10</b>	<b>How-tos</b>	<b>22</b>
	10.1 Extended page references (varioref) . . . . .	22
	10.2 \newtheorem . . . . .	23
	10.3 newfloat . . . . .	25
	10.4 Overriding the reference type . . . . .	26
	10.5 enumitem . . . . .	26
	10.6 zref-xr . . . . .	27
	10.7 tcolorbox . . . . .	28
	10.8 Ordinal references . . . . .	29
<b>11</b>	<b>\label or \zlabel?</b>	<b>30</b>
<b>12</b>	<b>Limitations</b>	<b>30</b>
<b>13</b>	<b>Compatibility modules</b>	<b>31</b>
<b>14</b>	<b>Work-arounds</b>	<b>33</b>
<b>15</b>	<b>Acknowledgments</b>	<b>35</b>
<b>16</b>	<b>Change history</b>	<b>36</b>

# 1 Introduction

Cross-referencing is an area which lends itself quite naturally to automation. Not only for input convenience but also, and most importantly, for end results consistency. Indeed, the standard  $\LaTeX$  cross-referencing system – with `\label`, `\ref`, and `\pageref` – is already a form of automation, by relieving us from checking the number of the referenced object, and the page where it lies.

But the plethora of existing features, packages and document classes which, in one way or another, extends this basic functionality is a clear indication of a demand for more automation. Just to name the some of the most popular: `cleveref`, `nameref`, `titleref`, `varioref`, `fancyref`, and the kernel’s `\labelformat`.

However, the standard cross-referencing system traditionally stored two, and only two, properties with the label: the printed representation of the counter last incremented with `\refstepcounter` and the page. Of course, out of the mentioned desire to automate more, the need arose to store more information about the label to support this: the title or caption of the referenced object; its counter or, even better, its “type”, that is, whether it is a section, the chapter, figure, etc.; its hyperlink anchor, and so on. Thus those two property “fields” of the standard label became quite a disputed real state. And the packages in this area of functionality, or which somehow depended on extending the cross-referencing mechanism, were bound to step on each other’s toes as a result. Nowadays (2024), many of these issues have changed for the better. Notably, now the kernel’s `\label` already stores five arguments, regardless of the presence of `hyperref`, and also provides a hook with the label as argument so that packages can extend its functionality without the need to redefine it, as `nameref` now already does (and so does `zref-clever`).<sup>1</sup>

Out of this conundrum, Heiko Oberdiek eventually developed `zref`, which implements an extensible referencing system, making the labels store a property list of flexible length, so that new properties can be easily added and queried. However, even when `zref` can rightfully boast this powerful basic concept and is really quite featureful, with several different modules available, it is fair to say that, for the average user, the package may appear to be somewhat raw. Indeed, for someone who “just wants to make a cross-reference”, the user interface of the `zref-user` module is akin to the standard  $\LaTeX$  cross-referencing system, and even requires some extra work if you want to have a hyperlinked reference. In other words, `zref` seems to have focused on infrastructure and on performing a number of specialized tasks with different modules, and a large part of the landscape of automation features available for the standard referencing system was not carried over to `zref`, neither by the `zref` itself nor by other packages.

`zref-clever` tries to cover this gap, by bringing a number of existing features available for the standard referencing system to `zref`. And the package’s name makes it clear that the core of the envisaged feature set is that of `cleveref`, even though the attempt was less one of replicating functionality per se than that of having it as a successful point of reference, from where we could then try to tap into `zref`’s potential. Indeed, although there is a significant intersection, the features of `zref-clever` are neither a superset nor a subset of those of `cleveref`. There are things either of them can do that the other can’t. There are also important differences in user interface design. In particular, `zref-clever` relies heavily on `key=value` interfaces both for general configuration and for centering in a single user command, `\zcref`, as the main entrance for reference making, whose behavior can be modulated by local options.

---

<sup>1</sup>We are currently observing some of the pains of adjusting the ecosystem to these improvements though...

Considering that zref itself offers the zref-titleref module, and that zref-vario offers integration of zref-clever with varioref, a significant part of the most prominent automation features available to the standard referencing system is thus brought to zref, working under a single consistent underlying infrastructure and user interface. All in all, hopefully zref-clever can make zref more accessible to the average user, and more interesting to users in general.

## 2 Warning

This package is in its early days, and should be considered experimental. By this I don't mean I expect it to be "edgy", indeed quite a lot of effort has been put into it so that this is not the case. However, during the initial development, I had to make a number of calls for which I felt I had insufficient information: in relation to features, packages, or classes I don't use much, or to languages I don't know well, user needs I found hard to anticipate etc. Hence, the package needs some time, and some use by more adventurous people, until it can settle down with more conviction. In the meantime, polishing the user interface and the infrastructure have a clear priority over backward compatibility. So, if you choose to use this package, you should be ready to accommodate to eventual upstream changes.

## 3 zref-clever for the impatient

Making cross-references with zref-clever is very similar to the standard referencing system in its user interface: one sets a label, with `\label`, which can then be referenced, in our case with `\zcref`, which is the main and basic entrance for making references with zref-clever. `\zcref` receives a list of labels as argument and its behavior can be locally changed through the optional argument.

A basic document using zref-clever is shown in How-to 1 which should feel very familiar to any L<sup>A</sup>T<sub>E</sub>X user acquainted with the standard referencing system:

### How-to 1: Basic usage

```
\documentclass{article}
\usepackage{zref-clever}
\usepackage{hyperref}
\begin{document}
\section{Section 1}
\label{sec:section-1}
\begin{figure}
  A figure.
  \caption{Figure 1}
  \label{fig:figure-1}
\end{figure}
A reference to \zcref{sec:section-1}. \zcref[S]{fig:figure-1} shows some
interesting information.
A page reference can be done with either \zcpageref{sec:section-1} or with
\zcref[page]{sec:section-1}.
A reference can also be made to multiple labels, as in \zcref{sec:section-1,
fig:figure-1}.
\end{document}
```

The references in the document of How-to 1 illustrate one of the main features of `zref-clever`, which is to include an appropriate “type name” of the reference, alongside of the reference itself. The document renders its references as:

A reference to section 1. Figure 1 shows some interesting information. A page reference can be done with either page 1 or with page 1. A reference can also be made to multiple labels, as in section 1 and figure 1.

How-to 1 also illustrates the use the optional argument of `\zcref` to modulate the behavior of the reference. In particular, the `S` option is one you should get acquainted with from the start, as it ensures that the type name of a reference is properly capitalized and not abbreviated, and it is meant to be used in references made at the beginning of a sentence.

But `zref-clever` is highly customizable, and several other options exist, and multiple places where you can set them:

- The optional argument of `\zcref`: for individual references.
- `\zcsetup`: for settings meant to affect all references.
- `\zcRefTypeSetup`: to customize the behavior of each reference type.
- `\zcLanguageSetup`: for language-specific settings.

For the list of available options, and for these different scopes in which they can be used, see Sections 5, 6, 8 and 9. Other usage examples are available at Section 10.

## 4 Loading the package

`zref-clever` can be loaded with the usual:

```
\usepackage{zref-clever}
```

The package does not accept load-time options, package options must be set using `\zcsetup` (see Section 5).

### 4.1 Dependencies

`zref-clever` requires `zref`, particularly its `zref-base`, `zref-user` and `zref-abspage` modules, and the  $\LaTeX$  kernel 2023-11-01, or newer. It requires UTF-8 input encoding, which has been the kernel’s default for some time. It also needs `ifdraft`. Some packages are leveraged by `zref-clever` if they are present, but are not loaded by default or required by it, namely: `hyperref`, `zref-check`, and `zref`’s `zref-hyperref` and `zref-xr` modules.

## 5 User interface

---

`\zcref` `\zcref` $\langle$  $\ast$  $\rangle$  $[\langle$  $options$  $\rangle]$  $\{\langle$  $labels$  $\rangle\}$

Typesets references to  $\langle labels \rangle$ , given as a comma separated list. When `hyperref` support is enabled, references will be hyperlinked to their respective anchors, according to options. The starred version of the command does the same as the plain one, just does

not form links. The `<options>` are (mostly) the same as those of the package, and can be given to local effect. The `<labels>` argument is protected by `zref's \zref@wrapper@babel`, so that it enjoys the same support for `babel's` active characters as `zref` itself does.

---

`\zcpageref` `\zcpageref{*}[<options>]{<labels>}`

Typesets page references to `<labels>`, given as a comma separated list. It is equivalent to calling `\zceref` with the `ref=page` option: `\zceref{*}[<options>,ref=page]{<labels>}`.

---

`\zcsetup` `\zcsetup{<options>}`

Sets `zref-clever's` general options (see Sections 6 and 8). The settings performed by `\zcsetup` are local, within the current group. But, of course, it can also be used to global effects if ungrouped, e.g. in the preamble.

---

`\zcRefTypeSetup` `\zcRefTypeSetup{<type>}{<options>}`

Sets type-specific reference format options (see Section 8). Just as for `\zcsetup`, the settings performed by `\zcRefTypeSetup` are local, within the current group.

Besides these, user facing commands related to Internationalization are presented in Section 9. Note still that all user commands are defined with `\NewDocumentCommand`, which translates into the usual handling of arguments by it and/or processing by `l3keys`, particularly with regard to brace-stripping and space-trimming.

Furthermore, `zref-clever` loads `zref's` `zref-user` module by default. So you also have its user commands available out of the box, including `\zref` and `\zpageref`, but notably:

## 6 Options

`zref-clever` is highly configurable, offering a lot of flexibility in typeset results of the references, but it also tries to keep these “handles” as convenient and user friendly as possible. To this end, most of what one can do with `zref-clever` (pretty much all of it), can be achieved directly through the standard and familiar “comma separated list of `key=value` options”.

There are two main groups of options in `zref-clever`: “general options”, which affect the overall behavior of the package, or the reference as a whole; and “reference format options”, which control the detail of reference formatting, including type-specific and language-specific settings.

This section covers the first group (for the second one, see Section 8). General options can be set globally by means of `\zcsetup` in the preamble (see Section 5). They can also be set locally with `\zcsetup` along the document or through the optional argument of `\zceref` (see Section 5). Most general options can be used in any of these contexts, but that is not necessarily true for all cases, some restrictions may apply, as described in each option’s documentation.

`ref`            The `ref` option controls the label property to which `\zceref` refers to. It can receive  
`page` `zref` properties, as long as they are declared, but notably `default`, `page`, `thecounter` and, if `zref-titleref` is loaded, `title`. The initial value is `default`, which is our standard reference. `thecounter` is a property set by `zref-clever` and is similar to `zref's` `default` property, except

that it is not affected by the kernel's `\labelformat`.<sup>2</sup> By default, reference formatting, sorting, and compression are done according to information inferred from the *current counter* (see `currentcounter` option below). Special treatment in these areas is provided for page, but not for any other properties. The `page` option is a convenience alias for `ref=page`.

<code>typeset</code> <code>noname</code> <code>noref</code>	<p>When <code>\zcref</code> typesets a set of references, each group of references of the same type can be, and by default are, preceded by the type's "name", and this is indeed an important feature of <code>zref-clever</code>. This is optional however, and the <code>typeset</code> option controls this behavior. It can receive values <code>ref</code>, in which case it typesets only the reference(s), <code>name</code>, in which case it typesets only the name(s), or <code>both</code>, in which case it typesets, well, both of them. Note that, when value <code>name</code> is used, the name is still typeset according to the set of references given to <code>\zcref</code>. For example, for multiple references, the plural form is used, capitalization options are honored, etc. Also hyperlinking behaves just <i>as if</i> the references were present and, depending on the corresponding options, the name may be linked to the first reference of the type group. The <code>noname</code> and <code>noref</code> options are convenience aliases for <code>typeset=ref</code> and <code>typeset=name</code>, respectively.</p>
<code>sort</code> <code>nosort</code>	<p>The <code>sort</code> option controls whether the list of <code>\langle labels \rangle</code> received as argument by <code>\zcref</code> should be sorted or not. It is a boolean option, with initial value <code>true</code>. The <code>nosort</code> option is a convenience alias for <code>sort=false</code>.</p>
<code>typesort</code> <code>notypesort</code>	<p>Sorting references of the same type can be done with well defined logical criteria. They either have the same counter or their counters share a clear hierarchical relation (in the resetting behavior), such that a definite sorting rule can be inferred from the label's data. The same is not true for sorting of references of different types. Should "tables" come before or after "figures"? The <code>typesort</code> option allows to specify the sorting priority of different reference types. It receives as value a comma separated list of reference types, specifying that their sorting is to be done in the order of that list. But <code>typesort</code> does not need to receive <i>all</i> possible reference types. The special value <code>{{othertypes}}</code> (yes, double braced, one for <code>l3keys</code>, so that the second can make the list) can be placed anywhere along the list, to specify the sort priority of any type not included explicitly. If <code>{othertypes}</code> is not present in the list, it is presumed to be at the end of it. Any unspecified types (that is, those falling implicitly or explicitly into the <code>{othertypes}</code> category) get sorted between themselves in the order of their first appearance in the label list given as argument to <code>\zcref</code>. I presume the common use cases will not need to specify <code>{othertypes}</code> at all but, for the sake of example, if you just really dislike equations, you could use <code>typesort={{othertypes}}, equation</code>. <code>typesort</code>'s default value is <code>{part, chapter, section, paragraph}</code>, which places the sectioning reference types first in the list, in their hierarchical order, and leaves everything else to the order of appearance of the labels. The <code>notypesort</code> option behaves like <code>typesort={{othertypes}}</code> would do, that is, it sorts all types in the order of the first appearance in the labels' list.</p>
<code>comp</code> <code>nocomp</code>	<p><code>\zcref</code> can automatically compress a set of references of the same type into a range, when they occur in immediate sequence. The <code>comp</code> controls whether this compression should take place or not. It is a boolean option, with initial value <code>true</code>. The <code>nocomp</code> option</p>

---

<sup>2</sup>Technical note: the default property stores `\@currentlabel`, while the `thecounter` property stores `\the\@currentcounter`. The later is exactly what `\refstepcounter` uses to build `\@currentlabel`, except for the `\labelformat` prefix and, hence, has the advantage of being unaffected by it. But the former is *more reliable* since `\@currentlabel` is expected to be correct pretty much anywhere whereas, although `\refstepcounter` does set `\@currentcounter`, it is not everywhere that uses `\refstepcounter` for the purpose. In the cases where the references from these two do diverge, `zref-clever` will likely misbehave (reference type, sorting and compression inevitably depend on a correct `currentcounter`), but using `default` at least ensures that the reference itself is correct. That said, if you do set `\labelformat` for some reason, `thecounter` may be useful.

is a convenience alias for `comp=false`. Of course, for better compression results the sort is recommended, but the two options are technically independent.

`endrange` The `endrange` option provides additional control over how the end reference of a range is typeset, so as to achieve terse ranges. The option can operate in two technically different ways. It may receive one of a number of predefined values, which can process the end reference of the range, comparing it with the beginning reference, to achieve a given end result.<sup>3</sup> Or, it can specify a label property to be used directly, without any processing. The available predefined values are: `ref`, `stripprefix`, `pagecomp`, and `pagecomp2`. `ref` corresponds to the default behavior, and instructs `\zcref` to use whatever property was set at the `ref` option for the end of range reference. `stripprefix` strips the common part at the start of each reference from the end one. `pagecomp` is the equivalent of `stripprefix` for page numbers, it does the same thing, but only if the references are comprised exclusively of Arabic numerals. `pagecomp2` is a variant of `pagecomp` that leaves at least two digits at the end reference (except for a leading zero). If values other than the predefined ones are given to `endrange` they are considered as label properties, as long as they are declared. This property is used to typeset the end of range reference if the label contains it, and if both references would be “compressible” according to the `comp` option, otherwise the property specified by the `ref` option is used. This is useful for things like sub-elements for which we can build a proper abbreviated sub-reference and populate the label with it (some compatibility modules already provide a number of such properties, but other ones can be built with `zref`, as needed).

`range` By default (that is, when the `range` option is not given), `\zcref` typesets a complete  
`rangetopair` list of references according to the `\labels` it received as argument, and only compresses some of them into ranges if the `comp` option is enabled and if references of the same type occur in immediate sequence. The `range` option makes `\zcref` behave differently. Sorting is implied by this option (the `sort` option is disregarded) and, for each reference type group in `\labels`, `\zcref` builds a range from the first to the last reference in it, even if references in between do not occur in immediate sequence. It is a boolean option, and the initial value is `false`. `\zcref` is smart enough to recognize when the first and last references of a type do happen to be contiguous, in which case it typesets a “pair”, instead of a “range”. But this behavior can be disabled by setting the `rangetopair` option to `false`.

`cap` The `cap` option controls whether the reference type names should be capitalized or  
`nocap` not. It is a boolean option, and it can also be set for specific reference types or languages  
`capfirst` (see Section 8). The `nocap` option is a convenience alias for `cap=false`. The `capfirst` option ensures that the reference type name of the *first* type block is capitalized, even when `cap` is set to `false`.

`abbrev` The `abbrev` option controls whether to use abbreviated reference type names when  
`noabbrev` they are available. It is a boolean option, and it can also be set for specific reference types  
`noabbrevfirst` or languages (see Section 8). The `noabbrev` option is a convenience alias for `abbrev=false`. The `noabbrevfirst` ensures that the reference type name of the *first* type block is never abbreviated, even when `abbrev` is set to `true`.

`S` `S` for “Sentence”. The `S` option is a convenience alias for `noabbrevfirst=true`, `capfirst=true`, and is intended to be used in references made at the beginning of a sentence. It is highly recommended that you make a habit of using the `S` option for

---

<sup>3</sup>For the  $\TeX$ nically inclined: those values that perform some processing – namely `stripprefix`, `pagecomp`, and `pagecomp2` – fully expand the references before comparing them, since it makes sense to perform this task as close as possible to the printed representation of the references. I don’t expect this to be a problem in normal use cases, but it does represent a limitation on what the references can contain. In case some control over this is needed, check the `zref-clever/endrange-setup` hook in the code documentation.



beginning of sentence references. Even if you do happen to be currently using `cap=true`, `abbrev=false`, proper semantic markup will ensure you get expected results even if you change your mind in that regard later on. For that reason, it was made short and mnemonic, it can't get any easier.

`hyperref` The `hyperref` option controls the use of `hyperref` by `zref-clever` and takes values `auto`, `true`, `false`. The initial value, `auto`, makes `zref-clever` use `hyperref` if it is loaded, meaning that references made with `\zcref` get hyperlinked to the anchors of their respective `\langle`labels`\rangle`. `true` does the same thing, but warns if `hyperref` is not loaded (`hyperref` is never loaded for you). In either of these cases, if `hyperref` is loaded, module `zref-hyperref` is also loaded by `zref-clever`. `false` means not to use `hyperref` regardless of its availability. This is a preamble only option, but `\zcref` provides granular control of hyperlinking by means of its starred version.

`nameinlink` The `nameinlink` option controls whether the type name should be included in the reference hyperlink or not (provided there is a link, of course). Naturally, the name can only be included in the link of the *first* reference of each type block. `nameinlink` can receive values `true`, `false`, `single`, and `tsingle`. When the value is `true` the type name is always included in the hyperlink. When it is `false` the type name is never included in the link. When the value is `single`, the type name is included in the link only if `\zcref` is typesetting a single reference (not necessarily having received a single label as argument, as they may have been compressed), otherwise, the name is left out of the link. When the value is `tsingle`, the type name is included in the link for each type block with a single reference, otherwise, it isn't. An example: suppose you make a couple of references to something like `\zcref{chap:chapter1}` and `\zcref{chap:chapter1, sec:section1, fig:figure1, fig:figure2}`. The "figure" type name will only be included in the hyperlink if `nameinlink` option is set to `true`. If it is set to `tsingle`, the first reference will include the name in the link for "chapter", as expected, but also in the second reference the "chapter" and "section" names will be included in their respective links, while that of "figure" will not. If the option is set to `single`, only the name for "chapter" in the first reference will be included in the link, while in the second reference none of them will. The initial value is `nameinlink=tsingle`, and the option given without a value is equivalent to `nameinlink=true`.


`lang` The `lang` option controls the language used by `\zcref` when looking for language-specific reference format options (see Section 8). The initial value, `current`, uses the current language, as defined by `babel` or `polyglossia` (or `english` if none of them is loaded). Value `main` uses the main document language, as defined by `babel` or `polyglossia` (or `english` if none of them is loaded). The `lang` option also accepts that the language be specified directly by its name, as long as it's a language known by `zref-clever`. For more details on Internationalization, see Section 9.

`d` The `d` option sets the declension case, and affects the type name used for typesetting the reference. Whether this option is operative, and which values it accepts, depends on the declared setup for each language. For details, see Section 9.


`nudge` This set of options revolving around `nudge` aims to offer some guard against mischievous automation on the part of `zref-clever` by providing a number of "nudges" (compilation time messages) for cases in which you may wish to revise material *surrounding* the reference – an article, a preposition – according to the reference typeset results. Useful mainly for languages which inflect the preceding article to gender and/or number, but may be used generally to fine-tune the language and style around the cross-references made with `\zcref`. The `nudge` option is the main entrance to this feature and takes values `true`, `false`, `ifdraft`, or `iffinal`. The first two, respectively, enable or disable the "nudging" unconditionally. With `ifdraft`, `nudge` keeps quiet when option `draft` is given

to `\documentclass`, while with `iffinal`, nudging is only enabled when option `final` is (explicitly) passed to `\documentclass`. The option given without a value is equivalent to `nudge=true` and the initial value is `nudge=false`. `nonudge` is a convenience alias for `nudge=false`, and can be used to silence individual references. The `nudgeif` option controls the events which may trigger a nudge. It takes a comma separated list of elements, and recognizes values `multitype`, `comptosing`, `gender`, and `all`. The `multitype nudge` warns when the reference is composed by multiple type blocks (see Section 8). The `comptosing nudge` let's you know when multiple labels of the same type have been compressed to a singular type name form. It can be combined with the `sg` option, which is the way to tell `\zcref` you know it's a singular and so not to nudge if a compression to singular occurs, but to nudge if the contrary occurs, that is, when a plural type name form is employed. The `gender nudge` must be combined with option `g`, and depends on the language having support for it. In essence language files can store the `gender(s)` of each type name (this is done for built-in language files, but can also be done with `\zclanguageSetup` for languages declared to support it). The `g` option let's you specify the gender you expect for that particular reference and the nudge is triggered if there is a mismatch between `g` and the `gender(s)` for the type name in the language file. Both the `comptosing` and the `gender nudes` have a type block as its scope. See Section 9 for more details and intended use cases of the “nudging” feature.


- font      The `font` option can receive font styling commands to change the appearance of the whole reference list (see also the `namefont` and `reffont` reference format options in Section 8). It does not affect the content of the note, however. The option is intended exclusively for commands that only change font attributes: `style`, `family`, `shape`, `weight`, `size`, `color`, etc. Anything else, particularly commands that may generate typeset output, is not supported.
- note      The `note` option receives as value some text to be typeset at the end of the whole reference list. It is separated from it by `notesep` (see Section 8).
- check     Provides integration of `zref-clever` with the `zref-check` package. The option is only functional in the document body and if `zref-check` has been loaded. `check` requires a value, which works exactly like the optional argument of `\zcheck`, and can receive both checks and `\zcheck`'s options. And the checks are performed for each label in  $\{\langle labels \rangle\}$  received as argument by `\zcref`. See the User manual of `zref-check` for details. The checks done by the `check` option in `\zcref` comprise the complete reference, including the note (see Section 8).
- countertype      The `countertype` option allows to specify the “reference type” of each counter, which is stored as a label property when the label is set. This reference type is what determines how a reference to this label will eventually be typeset when it is referred to (see Section 7). A value like `countertype = {foo = bar}` sets the `foo` counter to use the reference type `bar`. There's only need to specify the counter type for counters whose name differs from that of their type, since `zref-clever` presumes the type has the same name as the counter, unless otherwise specified. Also, the initial value of the option already sets appropriate types for basic L<sup>A</sup>T<sub>E</sub>X counters, including those from the standard classes. Setting a counter type to an empty value removes any (explicit) type association for that counter, in practice, this means it then uses a type equal to its name. The `reftype` option allows one to specify the reference type manually, regardless of the current counter. This can be used to locally override any `countertype` settings of the package and, for those acquainted with it, is the equivalent of `cleveref`'s optional argument to `\label`. Normally, you'd want to use this option within a group, but if you must do otherwise, the default value can be restored by setting the option without a value. Since these options only affect how labels are set, they are not available in `\zcref`.

 counterresetters  
counterresetby

The sorting and compression of references done by `\zcref` requires that we know the counter associated with a particular label but also information on any counter whose stepping may trigger its resetting, or its “enclosing counters”. This information is not easily retrievable from the counter itself but is (normally) stored with the counter that does the resetting. The `counterresetters` option receives a full comma separated list of counter names, which `zref-clever` uses to search for “enclosing counters” of the counter for which a label is being set. The order matters (see code documentation for details), and you should also take care to start from the initial value set by the package. Unfortunately, not every counter gets reset through the standard machinery for this, including some  $\LaTeX$  kernel ones (e.g. the `enumerate` environment counters). For those, there is really no way to retrieve this information directly, so we have to just tell `zref-clever` about them. And that’s what the `counterresetby` option is made for. It receives a comma separated list of `key=value` pairs, in which `key` is the counter, and `value` is its “enclosing counter”, that is, the counter whose stepping results in its resetting. This is not really an “option” in the sense of “user choice”, it is more of a way to inform `zref-clever` of something it cannot know or automatically find in general. One cannot place arbitrary information there, or `zref-clever` can be thoroughly confused. The setting must correspond to the actual resetting behavior of the involved counters. `counterresetby` has precedence over the search done in the `counterresetters` list. The initial value of `counterresetters` includes the counters for sectioning commands of the standard classes which, in most cases, should be the relevant ones for cross-referencing purposes. The initial value of `counterresetby` includes the `enumerate` environment counters. So, hopefully, you don’t need to ever bother with either of these options. But, if you do, they are here. Use them with caution though. Since these options only affect how labels are set, they are not available in `\zcref`.

 currentcounter

$\LaTeX$ ’s `\refstepcounter` sets two variables which potentially affect the `\label` set after it: `\@currentlabel` and `\@currentcounter`. But, since `zref-clever` relies heavily on the information of what the current counter is, it must set `zref` to store that information with the label, as it does. As long as the document element we are trying to refer to uses the standard machinery of `\refstepcounter` we are on solid ground and can retrieve the correct information. However, it is not always ensured that `\@currentcounter` is kept up to date. For example, packages which handle labels specially, for one reason or another, may or may not set `\@currentcounter` as required. Considering the addition of `\@currentcounter` to `\refstepcounter` itself is not that old,<sup>4</sup> it is likely that in some places a reliable `\@currentcounter` is not really in place. Therefore, it may happen we need to tell `zref-clever` what the current counter is in certain circumstances, and that’s what `currentcounter` does. The same as with the previous two options, this is not really an “user choice” kind of option, but a way to tell `zref-clever` a piece of information it has no means to retrieve automatically. The setting must correspond to the actual “current counter”, meaning here “the counter underlying `\@currentlabel`” in a given situation. Also, when using the `currentcounter` option, take care that the setting is duly grouped because, if set, it has precedence over `\@currentcounter` and, contrary to the later, the former is not reset the next time `\refstepcounter` runs. Its initial value is, quite naturally, `\@currentcounter`, which can be reset by calling the option with no value. Since this option only affects how labels are set, it is not available in `\zcref`.

 labelhook

`labelhook` is a boolean option which controls whether `zref-clever` uses the kernel’s label hook to, whenever a standard `\label` is called, also set a `\zlabel` with the same name (which is what `zref-clever` actually uses to make its references). The package’s initial

---

<sup>4</sup>Traditionally, only the current label was thus stored, the current counter was added to `\refstepcounter` somewhat recently (with the 2020-10-01 kernel release).

value is true so that one can use `\labels` along the document and refer to them (also with `zref` and `zref-clever` reference commands. This value is not only the default, but also a recommended value. Disabling this option means you are on your own to handle cases where `\zlabel` does not work, which are not many, but can be tricky to deal with where they occur. Since this option only affects how labels are set, it is not available in `\zceref`. See Section 11 for some discussion of different labeling approaches this options allows for.

nocompat

Some packages, document classes, or LaTeX features may require specific support to work with `zref-clever` (see Section 12). `zref-clever` tries to make things smoother by covering some of them. Depending on the case, this can take the form of some simple setup for `zref-clever`, or may involve the use of hooks to external environments or commands and, eventually, a patch or redefinition. By default, all the available compatibility modules are enabled. Should this be undesired or cause any problems in your environment, the option `nocompat` can selectively or completely inhibit their loading. `nocompat` receives a comma separated list of compatibility modules to disable (for the list of available modules and details about each of them, see Section 13). You can disable all modules by setting `nocompat` without a value (or an empty one). This is a preamble only option.

## 7 Reference types

A “reference type” is the basic `zref-clever` setup unit for specifying how a cross-reference group of a certain kind is to be typeset. Though, usually, it will have the same name as the underlying  $\LaTeX$  *counter*, they are conceptually different. `zref-clever` sets up *reference types* and an association between each *counter* and its *type*, it does not define the counters themselves, which are defined by your document. One *reference type* can be associated with one or more *counters*, and a *counter* can be associated with different *types* at different points in your document. But each label is stored with only one *type*, as specified by the counter-type association at the moment it is set, and that determines how the reference to that label is typeset. References to different *counters* of the same *type* are grouped together, and treated alike by `\zceref`. A *reference type* may be known to `zref-clever` when the *counter* it is associated with is not actually defined, and this inconsequential. In practice, the contrary may also happen, a *counter* may be defined but we have no *type* for it, but this must be handled by `zref-clever` as an error (at least, if we try to refer to it), usually a “missing name” error.

`zref-clever` provides initial settings for the following reference types: part, chapter, section, paragraph, appendix, subappendix, page, line, figure, table, item, footnote, endnote, note, equation, theorem, lemma, corollary, proposition, definition, proof, result, remark, example, algorithm, listing, exercise, and solution. Therefore, if you are using a language for which `zref-clever` has built-in support (see Section 9), these reference types are available for use out of the box.<sup>5</sup> And, in any case, it is always easy to setup custom reference types with `\zcRefTypeSetup` or `\zcLanguageSetup` (see Sections 5, 8 and 9).

The association of a *counter* to its *type* is controlled by the `countertype` option. As seen in its documentation, `zref-clever` presumes the *type* to be the same as the *counter* unless instructed otherwise by that option. This association, as determined by the local value of the option, affects how the `\label` is set, which stores the type among its properties.

<sup>5</sup>There may be slight availability differences depending on the language, but `zref-clever` strives to keep this complete list available for the languages it has built-in language files.

However, when it comes to typesetting, that is from the perspective of `\zcref`, only the *type* matters. In other words, how the reference is supposed to be typeset is determined at the point the `\label` gets set. In sum, they may be namesakes (or not), but type is type and counter is counter.

Indeed, a reference type can be associated with multiple counters because we may want to refer to different document elements, with different *counters*, as a single *type*, with a single name. One prominent case of this are sectioning commands. `\section`, `\subsection`, and `\subsubsection` have each their counter, but we'd like to refer to all of them by "sections" and group them together. The same for `\paragraph` and `\subparagraph`.

There are also cases in which we may want to use different *reference types* to refer to document objects sharing the same *counter*. Notably, the environments created with L<sup>A</sup>T<sub>E</sub>X's `\newtheorem` command and the `\appendix`.

One more observation about "reference types" is due here. A *type* is not really "defined" in the sense a variable or a function is. It is more of a "name" which `zref-clever` uses to look for a whole set of type-specific reference format options (see Section 8). Each of these options individually may be "set" or not, "defined" or not. And, depending on the setup and the relevant precedence rules for this, some of them may be required and some not. In practice, `zref-clever` uses the *type* to look for these options when it needs one, and issues a compilation warning when it cannot find a suitable value.

## 8 Reference format

Formatting how the reference is to be typeset is, quite naturally, a big part of the user interface of `zref-clever`. In this area, we tried to balance "flexibility" and "user friendliness". But the former does place a big toll overall, since there are indeed many places where tweaking may be desired, and the settings may depend on at least two important dimensions of variation: the reference type and the language. Combination of those necessarily makes for a large set of possibilities. Hence, the attempt here is to provide a rich set of "handles" for fine tuning the reference format but, at the same time, do not *require* detailed setup by the users, unless they really want it.

With that in mind, we have settled with a user interface for reference formatting which allows settings to be done in different scopes, with more or less overarching effects, and some precedence rules to regulate the relation of settings given in each of these scopes. There are four scopes in which reference formatting can be specified by the user, in the following precedence order: i) as *general options*; ii) as *type-specific options*; iii) as *language- and type-specific options*; and iv) as *language-specific default options*. Besides those, there's a fifth *internal* scope, with the least priority of all, a "fallback", for the cases where it is meaningful to provide some value, even for an unknown language. The package itself places the default setup for reference formatting at low precedence levels, and the users can easily and conveniently override them as desired.

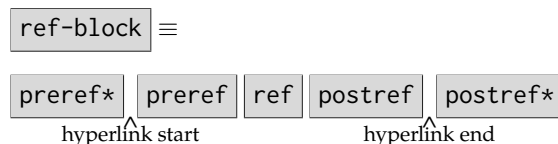
"General" options (i) can be given by the user in the optional argument of `\zcref`, but also set through `\zcsetup` (see Section 6). "Type" specific options (ii) are handled by `\zcRefTypeSetup` (see Section 5). "Language" options, whether "type" specific (iii) or "default" (iv) have their user interface in `\zcLanguageSetup`, and have their values populated by the package's built-in language files (see Section 9). Not all reference format specifications can be given in all of these scopes, though. Some of them can't be type-specific, others must be type-specific, so the set available in each scope depends on the

pertinence of the case. Table 1 introduces the available reference format options, which will be discussed in more detail soon, and lists the scopes in which each is available.

		General	Type	Language	
		(i)	(ii)	Type (iii)	Default (iv)
Typesetting (necessarily not type-specific)	<code>tpairsep</code>	•			•
	<code>tlistsep</code>	•			•
	<code>tlastsep</code>	•			•
	<code>notesep</code>	•			•
Typesetting (possibly type-specific)	<code>namesep</code>	•	•	•	•
	<code>pairsep</code>	•	•	•	•
	<code>listsep</code>	•	•	•	•
	<code>lastsep</code>	•	•	•	•
	<code>rangesep</code>	•	•	•	•
	<code>refbounds</code>	•	•	•	•
Typesetting (necessarily type-specific)	<code>Name-sg</code>		•	•	
	<code>name-sg</code>		•	•	
	<code>Name-pl</code>		•	•	
	<code>name-pl</code>		•	•	
	<code>Name-sg-ab</code>		•	•	
	<code>name-sg-ab</code>		•	•	
	<code>Name-pl-ab</code>		•	•	
<code>name-pl-ab</code>		•	•		
Font	<code>namefont</code>	•	•	•	•
	<code>reffont</code>	•	•	•	•
Other	<code>cap</code>	•	•	•	•
	<code>abbrev</code>	•	•	•	•
	<code>endrange</code>	•	•	•	•
	<code>rangetopair</code>	•	•	•	•

Table 1: Reference format options and their scopes

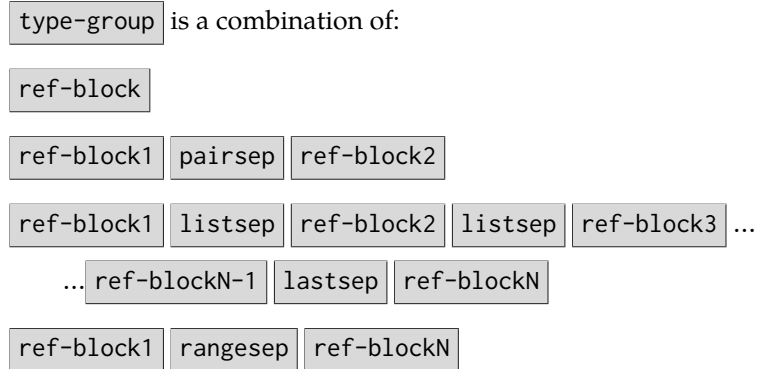
Understanding the role of each of these reference format options is likely eased by some visual schemes of how `zref-clever` builds a reference based on the labels' data and the value of these options. Take a `ref` to be that which a standard  $\LaTeX$  `\ref` would typeset. A `zref-clever` "reference block", or `ref-block`, is constructed as:



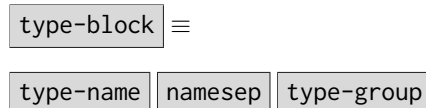
Where the `refbounds` option, which receives as value a comma separated list of four items in the form `{preref*,preref,postref,postref*}`, sets the surrounding elements to `ref`.<sup>6</sup> A `ref-block` is built for *each* label given as argument to `\zcref`. When the

<sup>6</sup>As usual, if each of the items contains start or end spaces, or commas anywhere, they must be protected

`<labels>` argument is comprised of multiple labels, each “reference type group”, or type-group is potentially made from the combination of single reference blocks, “reference block pairs”, “reference block lists”, or “reference block ranges”, where each is respectively built as:



To complete a “type-block”, a type-group only needs to be accompanied by the “type name”:



The type-name is determined not by one single reference format option but by the appropriate one among the `[Nn]ame-` options according to the composition of type-group and the general options. The reference format name options are eight in total: `Name-sg`, `name-sg`, `Name-pl`, `name-pl`, `Name-sg-ab`, `name-sg-ab`, `Name-pl-ab`, and `name-pl-ab`. The initial uppercase “N” signals the capitalized form of the type name. The `-sg` suffix stands for singular, while `-pl` for plural. The `-ab` is appended to the abbreviated type name form options. When setting up a type, not necessarily all forms need to be provided. `zref-clever` will always use the non-abbreviated form as a fallback to the abbreviated one, if the later is not available. Hence, if a reference type is not intended to be used with abbreviated names (the most common case), only the basic four forms are needed. Besides that, if you are using the `cap` option, only the capitalized forms will ever be required by `\zcref`, so you can get away setting only `Name-sg` and `Name-pl`. You should not do the contrary though, and provide only the non-capitalized forms because, even if you are using the `nocap` option, the capitalized forms will be still required for `capfirst` and `S` options to work. Whatever the case may be, you need not worry too much about being remiss in this area: if `\zcref` does lack a name form in any given reference, it will let you know with a compilation warning (and will typeset the usual missing reference sign: “??”).

A complete reference typeset by `\zcref` may be comprised of multiple type-blocks, in which case the “type-block-group” can also be made of single type blocks, “type block

---

by a pair of braces. However, care is taken that empty items don’t need such protection. So you can set, for example, something like `refbounds={ ( , , ) }` to get parentheses around your references, outside the hyperlink.

pairs” or “type block lists”, where each is respectively built as:

`type-block-group` is one of:

`type-block`

`type-block1` `tpairsep` `type-block2`

`type-block1` `tlistsep` `type-block2` `tlistsep` `type-block3` ...

... `type-blockN-1` `tlastsep` `type-blockN`

Finally, since `\zcref` can also receive an optional note, its full typeset output is built as:

A complete `\zcref` reference:

`type-block-group` `notesep` `note`

Reference format options can yet be divided in three general categories: i) “typesetting” options, the ones which we have seen thus far, as “building blocks” of the reference; ii) “font” options, which control font attributes of parts of the reference, namely `namefont` and `reffont`; and iii) “other” options. “Typesetting” options are intended exclusively for typesetting material: things you expect to see in the output of your references. “Font” options set the font, respectively, for the type-name and for ref (to set the font for the whole reference, see the `font` option in Section 6). These options are intended exclusively for commands that only change font attributes: style, family, shape, weight, size, color, etc. In either case, anything other than their intended uses is not supported.

Finally, a comment about the internal “fallback” reference format values mentioned above. These “last resort” option values are required by `zref-clever` for a clear particular case: if the user loads either `babel` or `polyglossia`, or explicitly sets a language, with a language that `zref-clever` does not know and has no language file for, it cannot guess what language that is, and thus has to provide some reasonable “language agnostic” default, at least for the options for which this makes sense. Users do not need to have access to this scope, since they know the language of their document, or know the values they want for those options, and can set them as general options, type-specific options, or language options through the user interface provided for the purpose. But the “fallback” options are documented here so that you can recognize when you are getting these values and change them appropriately as desired. Though hopefully reasonable, they may not be what you want. The “fallback” option values are the following:

```
tpairsep = {, _} ,
tlistsep = {, _} ,
tlastsep = {, _} ,
notesep  = { _} ,
namesep  = {\nobreakspace} ,
pairsep  = {, _} ,
listsep  = {, _} ,
lastsep  = {, _} ,
rangesep = {\textendash} ,
```



		General	Type	Language	
		(i)	(ii)	Type (iii)	Default (iv)
Base options	refbounds-first	•	•	•	•
	refbounds-first-sg	•	•	•	•
	refbounds-first-pb	•	•	•	•
	refbounds-first-rb	•	•	•	•
	refbounds-mid	•	•	•	•
	refbounds-mid-rb	•	•	•	•
	refbounds-mid-re	•	•	•	•
	refbounds-last	•	•	•	•
	refbounds-last-pe	•	•	•	•
	refbounds-last-re	•	•	•	•
Derived options (groups)	+refbounds-first	•	•	•	•
	+refbounds-mid	•	•	•	•
	+refbounds-last	•	•	•	•
	+refbounds-rb	•	•	•	•
	+refbounds-re	•	•	•	•
	+refbounds	•	•	•	•

Table 2: Advanced reference format options and their scopes

## 8.1 Advanced reference formatting

The reference format options discussed above and presented in Table 1 should suffice for most needs. However, if more fine-grained control of the reference format is needed, this can be achieved through a more detailed specification of `refbounds` for the different cases in which they may occur when a reference is processed. The options available for this purpose are presented in Table 2.

The “base” options are the actually operative ones, while the “derived” options are convenience aliases to set multiple base options in one go. In the naming scheme of these options, as is easy to presume, “first” refers to the first reference of a type-block, “mid” to the middle ones, and “last” to the last reference of the type-block. Less obviously, but hopefully still mnemonic enough, “sg” stands for “single”, “pb” and “pe” for “pair begin” and “pair end”, and finally “rb” and “re” for “range begin” and “range end”. Each of them receives as value a comma separated list of four items in the form {`preref*`, `preref`, `postref`, `postref*`}, just like `refbounds`.

The base options are mutually exclusive, which means, for example, that it is not sufficient to set `refbounds-first` to define the behavior of all first references of a type block. `refbounds-first` is the value used for the first reference when not single, not the beginning of a pair, and not the beginning of a range. Setting a group of them is the purpose of the derived options. Each of these sets all options under it. Some examples. `+refbounds-first` sets `refbounds-first`, `refbounds-first-sg`, `refbounds-first-pb`, and `refbounds-first-rb`. In turn, `+refbounds-rb` sets `refbounds-first-rb` and `refbounds-mid-rb`. And quite conveniently, `+refbounds` sets `+refbounds-first`, `+refbounds-mid`, and `+refbounds-last`, it is hence sufficient to set it to define the behavior of what is typeset around all references for the whole type-block. As you probably guessed by now, the `refbounds` option presented in Table 1 is an alias of `+refbounds`.

Language	Aliases	Language	Aliases
dutch		german	ngerman
english	american		austrian
	australian		naustrian
	british		swissgerman
	canadian		nswissgerman
	newzealand	italian	
	UKenglish	portuguese	brazilian
	USenglish		brazil
french	acadian		portuges
spanish		russian	

Table 3: Declared languages and aliases

Given that base and derived options are actually setting the same group of underlying options (the base ones), the order in which they are given is relevant: the last one prevails. The idea here is to use first the derived options to set some general defaults, and then change one or another base option to handle exceptions as needed. Of course, how best to use them depends on the case.

## 9 Internationalization

zref-clever provides internationalization facilities and integrates with babel and polyglossia to adapt to the languages in use by either of these language packages, or to a language specified directly by the user. This is primarily relevant for reference format options, particularly reference type *names* (though not only, since most reference format options can have language-specific values see Section 8). But other features of the package also cater for language specific needs.

As far as language selection is concerned, if the language is declared and zref-clever has a built-in “language file” for it, most use cases will likely be covered by the lang option (see Section 6), and its values `current` and `main`. When the lang option is set to `current` or `main`, zref-check will use, respectively, the *current* or *main* language of the document, as defined by babel or polyglossia.<sup>7</sup> Users can also set lang to a specific language directly, in which case babel and polyglossia are disregarded. zref-clever provides a number of built-in “language files”, for the languages listed in Table 3, which also includes the declared aliases to those languages.

zref-clever’s “language files” are loaded sparingly and lazily. A language file for a single language – that specified by user options in the preamble, which by default is the current document language – is loaded at `begindocument`. If any other language file is needed, it is loaded on the fly, if and when required. Of course, in either case, conditioned on availability. In sum, zref-clever loads as little as possible, but allows for convenient on the fly loading of language files if the values are indeed required, without users having to worry about it at all.

<sup>7</sup>Technically, zref-clever uses `\languagename` and `\bbl@main@language` for babel, and `\babelname` and `\mainbabelname` for polyglossia, which boils down to zref-clever always using *babel names* internally, regardless of which language package is in use. Indeed, an acquainted user will note that Table 3 contains only babel language names.

Language	declension	gender	allcaps
dutch	--	f,m,n	--
english	--	--	--
french	--	f,m	--
german	N,A,D,G	f,m,n	yes
italian	--	f,m	--
portuguese	--	f,m	--
spanish	--	f,m	--
russian	n,a,g,d,i,p	f,m,n	--

Table 4: Options for declared languages

But if the built-in language files do not cover your language, or if you'd like to adjust some of the default language-specific options, this can be done with `\zcDeclareLanguage`, `\zcDeclareLanguageAlias`, and `\zcLanguageSetup`.<sup>8</sup>

---

`\zcDeclareLanguage`    `\zcDeclareLanguage[⟨options⟩]{⟨language⟩}`

Declare a new language for use with `zref-clever`. If `⟨language⟩` has already been declared, just warn. The `⟨options⟩` argument receives the usual key=value list and recognizes three keys: `declension`, `gender`, and `allcaps`. `declension` receives a comma separated list of valid declension cases for `⟨language⟩`. The first element of the list is considered to be the default case, both for the `d` option in `\zceref` and for the `case` option in `\zcLanguageSetup`. Similarly, `gender` receives a comma separated list of genders for `⟨language⟩`. The elements in this list are those which are recognized as valid for the language for both the `g` option in `\zceref` and the `gender` option in `\zcLanguageSetup`. There is no default presumed in this case. Finally, `allcaps` can be used with languages for which nouns must be always capitalized for grammatical reasons. For a language declared with the `allcaps` option, the `cap` reference option (see Section 6) is disregarded, and `\zceref` always uses the capitalized type name forms. This means that language files for languages with such a trait can be halved in size, and that user customization for them is simplified, only requiring the capitalized name forms. On the other hand, the non-capitalized name- reference format options are rendered no-op for the language in question. Table 4 presents an overview of the options in effect for the languages declared by `zref-clever`. `\zcDeclareLanguage` is preamble only.

---

`\zcDeclareLanguageAlias`    `\zcDeclareLanguageAlias{⟨language alias⟩}{⟨aliased language⟩}`

Declare `⟨language alias⟩` to be an alias of `⟨aliased language⟩`. `⟨aliased language⟩` must be already known to `zref-clever`. Once set, the `⟨language alias⟩` is treated by `zref-clever` as completely equivalent to the `⟨aliased language⟩` for any language specification by the user. `\zcDeclareLanguageAlias` is preamble only.

---

`\zcLanguageSetup`    `\zcLanguageSetup{⟨language⟩}{⟨options⟩}`

Sets language-specific reference format options for `⟨language⟩` (see Section 8), be they type-specific or not. `⟨language⟩` must be already known to `zref-clever`. Besides reference

<sup>8</sup>Needless to say, if you'd like to contribute a language file or improve an existing one, that is much welcome at <https://github.com/gusbrs/zref-clever/issues>.

format options, `\zcLanguageSetup` knows three other keys: `type`, `case`, and `gender`. The first two work like a “switch” affecting the options *following* it. For example, if `type=foo` is given in `\zcOptions` the options following it will be set as type-specific options for reference type `foo`. Similarly, after `case=X` (provided `X` is a valid declension case for `\zcLanguage`), the following `[Nn]ame-` options will set values for the `X` declension case (other reference format options are not affected by case). Before the first occurrence of either `type` or `case` default values are set. For `case` this means the default declension case, which is the first element of the list provided to the `declension` option in `\zcDeclareLanguage`. For `type` this means language-specific but not type-specific option values (see Section 8). An empty valued `type=` key can also “unset” the type. The `gender` key sets the gender of the current type (provided the value it receives is one of the declared genders for `\zcLanguage`). For types which have multiple valid genders for a given language, the option can also receive a comma separated list. `\zcLanguageSetup` is preamble only.

A couple of examples to illustrate the syntax of `\zcLanguageSetup`:

```
\zcLanguageSetup{french}{
  type = section ,
  gender = f ,
  Name-sg = Section ,
  name-sg = section ,
  Name-pl = Sections ,
  name-pl = sections ,
}
\zcLanguageSetup{german}{
  type = section ,
  gender = m ,
  case = N ,
  Name-sg = Abschnitt ,
  Name-pl = Abschnitte ,
  case = A ,
  Name-sg = Abschnitt ,
  Name-pl = Abschnitte ,
  case = D ,
  Name-sg = Abschnitt ,
  Name-pl = Abschnitten ,
  case = G ,
  Name-sg = Abschnitts ,
  Name-pl = Abschnitte ,
}
```

As already noted, `zref-clever` has some support for languages with declension. This means mainly the declension of *nouns*, which is used for the reference type names. But some tools are also provided to support the user in getting better results for the text surrounding a reference, particularly for numbered and gendered articles, even if those don’t have their typeset output automated.

For reference type names, the declension cases for each language must be declared with `\zcDeclareLanguage`, and the name reference format options must be provided for each case, which is done for built-in language files of languages which have noun declension, and can be done by the user with `\zcLanguageSetup`, as we’ve seen. `zref-clever` does not try to guess or infer the case though, you must tell it to `\zceref`. And this is done by means of the `d` option (see Section 6). So you may write something like “nach den

`\zcref[d=D]{sec:section-1,sec:section-2}`” to get “nach den Abschnitten 1 und 2”. Or “`trotz des \zcref[d=G]{eq:theorem-1}`” to get “trotz des Theorems 1”.

Regarding the text surrounding the reference – the inflected article, the passing preposition, etc. –, the issue is more delicate. `zref-clever` cannot and intends not to typeset those for you. But, depending on the language, it is true that the kind of automation provided by `zref-clever` may betray your best efforts to get a proper surrounding text. Multiple labels passed to `\zcref` may result in singular type names, either because the labels are of different types, or because they got compressed into a single reference. References comprised of multiple type blocks may have each a name with a different gender. Or, worse, `tpairsep`, `tpairsep`, and `tlastsep` may not provide a general enough way to separate different type blocks in your language altogether. You may change something in your document that causes a label to change its type, and hence the gender of the type name. A page reference to a couple of floats which were by chance on the same page and all of a sudden no longer are. And so on.

In this area, the approach taken by `zref-clever` is to identify some typical situations in which your attention may be required in reviewing the surrounding text, and signal it at compilation time. Just like bad boxes, for example. This feature can be enabled by the `nudge` option (which is opt-in, see Section 6). There are three “nudges” available for this purpose which trigger messages at different events: `multitype`, `comptosing`, and `gender`. `multitype` nudges when a reference is comprised of multiple type blocks. `comptosing` when multiple labels of the same type block have been compressed into a single one and, hence, the type name used is singular. Finally, `gender` nudges when there is a mismatch between the gender specified in `\zcref` with the `g` option and the gender of the type name, as set in the language file or with `\zclanguageSetup`, for each type block. Which nudges to use is configurable with the option `nudgeif`. And, if you’re sure of the results for a particular `\zcref` call, you can always silence the nudges locally with the `nonudge` option.

The main reason to watch for multiple type references with the `multitype` nudge is that bundling together automatically a list of type blocks is less smooth an operation than it is for a single reference type. While it arguably works reasonably well for English, even there it is not always flawless, and depending on the language, results may range from “poor style” to outright wrong. A typical case would be of that of a language with inflected articles and a reference with multiple types of different genders or numbers. For example, in French, with a standard “`au \zcref{cha:chapter-3, sec:section-3.1}`” we get “au chapitre 3 et section 3.1” which sounds ugly, at best. So we may be better off writing instead “`au \zcref{cha:chapter-3} et à la \zcref{sec:section-3.1}`”. Or something else, of course. But the general point is that, depending on circumstances and on the language, the results of automating the grouping of multiple reference types, as `zref-clever` is able to do, may leave things to be desired for. Hence it lets you know when one such case occurs, so that you can review it for best results.

The case of the `comptosing` and `gender` nudges is more objective in nature, they respectively signal mismatches of number and gender. When a reference is made with `\zcref` to a single label we are sure the type name will be a singular form. However, when `\zcref` receives multiple labels of the same type, the type name will normally be a plural, but not necessarily so, since the labels may be compressed into a single one (see the `comp` option in Section 6), in which case the singular is used. The compression of multiple labels into a single reference should be an exception for default references, but not so for page references, where it is easy to conceive practical situations where it may occur. Suppose, for example, you have two contiguous floats in your document and make a page reference to both of them. Will they end up in the same page or not? Maybe we

know what the current state is, but we cannot know what may happen as the document keeps being edited. As a consequence, we don't know whether that reference will end up having a plural or a singular type name. That given, the logic of the `comptosing` nudge is the following. If we are giving multiple labels to `\zcref`, we can *presume* a plural type name, but we get a nudge in case the compression of the labels results in a singular type name form. If one such compression did happen to one of your references, you can use a singular article and then tell `\zcref` you did so with option `sg`. The effect of the `sg` option is to inhibit the nudge when a compression to singular occurs, but to do it instead when the compression *ceases* to occur, that is, if we get a plural type name again at some point.

The gender nudge aims to guard against one particular situation: possible changes of a reference's type. This does not occur by reason of any internal behavior of `zref-clever`, but it may be caused by changes in the document. You may wish to change one theorem into a proposition and, if you're writing in French or Portuguese, for example, that implies that the reference to it changes gender and the likely preceding article will no longer pass to the reference. The gender nudge requires that the gender of each type name and of each reference be explicitly specified. For the type names, this is done for the built-in language files of languages where this matters, and can be done with `\zLanguageSetup` as well. For the references, that is the purpose of the `g` option. When there is a mismatch between the two for any type block, the nudge is triggered. Of course, this means that the gender markup has to be supplied in the document at each reference. And given such type changes may not be frequent for you, or considered not particularly problematic, you'll have to balance if doing so is worth it. Still, the feature is available, and it's up to you.

## 10 How-tos

This section gathers some usage examples, or “how-tos”, of cases which may require some `zref-clever` setup, or usage adjustments, and each item is set around a cross-reference “task” we'd like to perform with `zref-clever`.

### 10.1 Extended page references (`varioref`)

**Task** Make cross-references to pages which are sensitive to the relative position between the reference and the label being referred to using `varioref`.

The `zref-vario` package offers a layer of compatibility with `varioref` and provides `\z...` counterparts for the latter's main reference commands.

How-to 2: `zref-vario`

```
\documentclass{article}
\usepackage{zref-clever}
\usepackage{zref-vario}
\begin{document}
\section{Section 1}
\label{sec:section-1}
\begin{figure}
  A figure.
  \caption{Figure 1}
  \label{fig:figure-1}
\end{figure}
```

```

\begin{figure}
  Another figure.
  \caption{Figure 2}
  \label{fig:figure-2}
\end{figure}
\zhref[S]{sec:section-1}\par
\zvpageref{fig:figure-1}\par
\zvrefrange{fig:figure-1}{fig:figure-2}\par
\zvpagerefrange{fig:figure-1}{fig:figure-2}\par
\zfullref{fig:figure-1}
\end{document}

```

## 10.2 \newtheorem

Since L<sup>A</sup>T<sub>E</sub>X's `\newtheorem` allows users to create arbitrary numbered environments, with respective arbitrary counters, the most `zref-clever` can do in this regard is to provide some “typical” built-in reference types to smooth user setup but, in the general case, some user setup may be indeed required. The examples below are equally valid for `amsthm`'s `\newtheorem` since, even if it provides features beyond those available in the kernel, its syntax and underlying relation with counters is pretty much the same. The same for `ntheorem`. For `thmtools`' `\declaretheorem`, though some adjustments to the examples below may be required, the basic logic is the same (there is no integration with the `Refname`, `refname`, and `label` options, which are targeted to the standard reference system, but you don't actually need them to get things working conveniently).

### Simple case

**Task** Setup up a new theorem environment created with `\newtheorem` to be referred to with `\zcref`. The theorem environment does not share its counter with other theorem environments, and one of `zref-clever` built-in reference types is adequate for my needs.

Suppose you set a “Lemma” environment with:

```
\newtheorem{lemma}{Lemma}[section]
```

In this case, since `zref-clever` provides a built-in `lemma` type (for supported languages) and presumes the reference type to be the same name as the counter, there is no need for setup, and things just work out of the box. So, you can go ahead with:

How-to 3: `\newtheorem`, simple case

```

\documentclass{article}
\usepackage{zref-clever}
\newtheorem{lemma}{Lemma}[section]
\begin{document}
\section{Section 1}
\begin{lemma}\label{lemma-1}
  A lemma.
\end{lemma}
\zcref{lemma-1}
\end{document}

```

If, however, you had chosen an environment name which did not happen to coincide with the built-in reference type, all you'd need to do is instruct `zref-clever` to associate the counter for your environment to the desired type with the `countertype` option:

#### How-to 4: `\newtheorem`, simple case

```
\documentclass{article}
\usepackage{zref-clever}
\zcsetup{countertype={lem=lemma}}
\newtheorem{lem}{Lemma}[section]
\begin{document}
\section{Section 1}
\begin{lem}\label{lemma-1}
  A lemma.
\end{lem}
\zcref{lemma-1}
\end{document}
```

### Shared counter

**Task** Setup up two new theorem environments created with `\newtheorem` to be referred to with `\zcref`. The theorem environments share the same counter, and the available `zref-clever` built-in reference types are adequate for my needs.

In this case, we need to set the `countertype` option in the appropriate contexts, so that the labels of each environment get set with the expected reference type. As we've seen (at Section 5), `\zcsetup` has local effects, so it can be issued inside the respective environments for the purpose. Even better, we can leverage the kernel's new hook management system and just set it for all occurrences with `\AddToHook{env/<myenv>/begin}`.

#### How-to 5: `\newtheorem`, shared counter

```
\documentclass{article}
\usepackage{zref-clever}
\AddToHook{env/mytheorem/begin}{%
  \zcsetup{countertype={mytheorem=theorem}}}
\AddToHook{env/myproposition/begin}{%
  \zcsetup{countertype={mytheorem=proposition}}}
\newtheorem{mytheorem}{Theorem}[section]
\newtheorem{myproposition}[mytheorem]{Proposition}
\begin{document}
\section{Section 1}
\begin{mytheorem}\label{theorem-1}
  A theorem.
\end{mytheorem}
\begin{myproposition}\label{proposition-1}
  A proposition.
\end{myproposition}
\zcref{theorem-1, proposition-1}
\end{document}
```



## Custom type

**Task** Setup up a new theorem environment created with `\newtheorem` to be referred to with `\zceref`. The theorem environment does not share its counter with other theorem environments, but none of `zref-clever` built-in reference types is adequate for my needs.

In this case, we need to provide `zref-clever` with settings pertaining to the custom reference type we'd like to use. Unless you need to typeset your cross-references in multiple languages, in which case you'd require `\zcLanguageSetup`, the most convenient way to setup a reference type is `\zcRefTypeSetup`. In most cases, what we really need to provide for a custom type are the "type names" and other reference format options can rely on default language options already provided by the package (assuming the language is supported).

How-to 6: `\newtheorem`, custom type

```
\documentclass{article}
\usepackage{zref-clever}
\newtheorem{myconjecture}{Conjecture}[section]
\zcRefTypeSetup{myconjecture}{
  Name-sg = Conjecture ,
  name-sg = conjecture ,
  Name-pl = Conjectures ,
  name-pl = conjectures ,
}
\begin{document}
\section{Section 1}
\begin{myconjecture}\label{conjecture-1}
  A conjecture.
\end{myconjecture}
\zceref{conjecture-1}
\end{document}
```

## 10.3 newfloat

**Task** Setup a new float environment created with `newfloat` to be referred to with `\zceref`. None of `zref-clever` built-in reference types is adequate for my needs.

The case here is pretty much the same as that for `\newtheorem` with a custom type. Hence, we need to setup a corresponding type, for which providing the "type names" should normally suffice. Note that, as far as `zref-clever` is concerned, there's nothing specific to the `newfloat` package in the setup, the same procedure can be used with `memoir's` `\newfloat` command or with the `float`, `floatrow`, and `trivfloat` packages.

How-to 7: `newfloat`

```
\documentclass{article}
\usepackage{newfloat}
\DeclareFloatingEnvironment{diagram}
\usepackage{zref-clever}
\zcRefTypeSetup{diagram}{
  Name-sg = Diagram ,
  name-sg = diagram ,
}
```

```

    Name-pl = Diagrams ,
    name-pl = diagrams ,
}
\begin{document}
\section{Section 1}
\begin{diagram}
  A diagram.
  \caption{A diagram}
  \label{diagram-1}
\end{diagram}
\zcref{diagram-1}
\end{document}

```

## 10.4 Overriding the reference type

**Task** Make references to a system of equations, which should be referred to in the plural.

Though, usually, setting the `countertype` option may provide a more general and convenient way to set the reference type of a given counter, sometimes we just need to do it for a particular label or set of labels. The `reftype` option allows us to do so and set the reference type directly, regardless of what the current counter is.

How-to 8: Overriding the reference type

```

\documentclass{article}
\usepackage{amsmath}
\usepackage{zref-clever}
\usepackage{hyperref}
\zcRefTypeSetup{pluralequation}{
  Name-sg = Equations ,
  name-sg = equations ,
  Name-pl = Equations ,
  name-pl = equations ,
}
\begin{document}
\section{Section 1}
\begin{equation}
  \zcsetup{reftype=pluralequation}
  \label{eq:1}
  \begin{aligned}
    A+B&=B+A\\
    C&=D+E\\
    E&=F
  \end{aligned}
\end{equation}
\zcref{eq:1}
\end{document}

```

## 10.5 enumitem

**Task** Setup a custom enumerate environment created with `enumitem` to be referred to.

Since the enumerate environment's counters are reset at each nesting level, but not with the standard machinery, we have to inform zref-clever of this resetting behavior with the counterresetby option. Also, given the naming of the underlying counters is tied with the environment's name and the level's number, we cannot really rely on an implicit counter-type association, and have to set it explicitly with the countertype option.

#### How-to 9: enumitem

```

\documentclass{article}
\usepackage{zref-clever}
\zcsetup{
  countertype = {
    myenumeratei = item ,
    myenumerateii = item ,
    myenumerateiii = item ,
    myenumerateiv = item ,
  } ,
  counterresetby = {
    myenumerateii = myenumeratei ,
    myenumerateiii = myenumerateii ,
    myenumerateiv = myenumerateiii ,
  }
}
\usepackage{enumitem}
\newlist{myenumerate}{enumerate}{4}
\setlist[myenumerate,1]{label=(\arabic*)}
\setlist[myenumerate,2]{label=(\Roman*)}
\setlist[myenumerate,3]{label=(\Alph*)}
\setlist[myenumerate,4]{label=(\roman*)}
\begin{document}
\begin{myenumerate}
\item An item.\label{item-1}
\begin{myenumerate}
\item An item.\label{item-2}
\begin{myenumerate}
\item An item.\label{item-3}
\begin{myenumerate}
\item An item.\label{item-4}
\end{myenumerate}
\end{myenumerate}
\end{myenumerate}
\end{myenumerate}
\zcref{item-1, item-2, item-3, item-4}
\end{document}

```

## 10.6 zref-xr

**Task** Make references to labels set in an external document.

zref itself offers this functionality with module zref-xr, and zref-clever is prepared to make use of it. Just a couple of details have to be taken care of, for it to work as intended: i) zref-clever must be loaded in both the main document and the external document, so that the imported labels also contain the properties required by zref-clever;

ii) since `\xexternaldocument` defines any properties it finds in the labels from the external document when it imports them, it must be called after `zref-clever` is loaded, otherwise the later will find its own internal properties already defined when it does get loaded, and will justifiably complain. Note as well that the starred version of `\xexternaldocument*`, which imports the standard labels from the external document, is not sufficient for `zref-clever`, since the imported labels will not contain all the required properties.

Assuming here `documentA.tex` as the main file and `documentB.tex` as the external one, and also assuming we just want to refer in “A” to the labels from “B”, and not the contrary, a minimum setup would be the following.

#### How-to 10: zref-xr

`documentA.tex`:

```
\documentclass{article}
\usepackage{zref-clever}
\usepackage{zref-xr}
\xexternaldocument[B-]{documentB}
\usepackage{hyperref}
\begin{document}
\section{Section A1}
\label{sec:section-a1}
\zcref{sec:section-a1, B-sec:section-b1}
\end{document}
```

`documentB.tex`:

```
\documentclass{article}
\usepackage{zref-clever}
\usepackage{hyperref}
\begin{document}
\section{Section B1}
\label{sec:section-b1}
\end{document}
```

## 10.7 tcolorbox

**Task** Make references to boxes from the `tcolorbox` package.

`tcolorbox` has support for `zref` and `zref-clever`, hence setting labels with the `label` option works out of the box (as long as `labelhook` is enabled). If you are using the `auto counter`, or some other custom counter, `tcolorbox`’s `label` type option can be used to set `zref-clever`’s `reftype` for the relevant scope. Furthermore, those who prefer to work with `\zlabels` only can set the `label` is `zlabel` option, which makes the `label` option set a `\zlabel` directly.

#### How-to 11: tcolorbox

```
\documentclass{article}
\usepackage{zref-clever}
\usepackage{zref-titleref}
\usepackage{tcolorbox}
\tcbuselibrary{theorems}
\usepackage{hyperref}
\newtcolorbox[auto counter,number within=section]{pabox}[2][]{%
```

```

    label type=example,
    title=Example~\thetcbcounter: #2,#1}
\newtcolorbox[use counter from=pabox,number within=section]{pabox2}[2][]{%
  label type=solution,
  title=Solution~\thetcbcounter: #2,#1}
\newtcbtheorem[number within=section]{mytheo}{My Theorem}{%
  label type=mytheorem}{th}
\zcrefTypeSetup{mytheorem}{
  Name-sg=Mytheorem,
  name-sg=mytheorem,
  Name-pl=Mytheorems,
  name-pl=mytheorems,
}
\begin{document}
\section{Section 1}
\label{sec:section-1}
\begin{pabox}[label={box:1}]{Title text}
  This is tcolorbox \zcref{box:1} on \zcpageref{box:1}.
\end{pabox}
\begin{pabox2}[label={box:2}]{Title text}
  This is tcolorbox \zcref{box:2} on \zcpageref{box:2}.
\end{pabox2}
\begin{mytheo}{This is my title}{theo}
  This is \zcref{th:theo} on \zcpageref{th:theo} and it is titled
  ``\zcref[noname,ref=title]{th:theo}``.
\end{mytheo}
\end{document}

```

## 10.8 Ordinal references

**Task** Typesetting the references as ordinal numbers.

This example is intended as an illustration of the flexibility `zref`'s extensible referencing system grants us.<sup>9</sup> Getting references as ordinal numbers, something that would be normally a tricky task, can be handled with a simple custom `zref` property, which we then use to set `zref-clever`'s `ref` option.

How-to 12: Ordinal references

```

\documentclass{article}
\usepackage{zref-clever}
\usepackage{fmtcount}
\makeatletter
\zref@newprop{ordref}{\ordinal{\@currentcounter}}
\zref@addprop{main}{ordref}
\makeatother
\zcsetup{ref=ordref}
\begin{document}
\section{Section 1}
\label{sec:section-1}
\begin{figure}

```

---

<sup>9</sup>Though clearly simplified, the example is less than academic. See <https://tex.stackexchange.com/a/670088> for an application to add a reference suffix needed in Turkish.

```

A figure.
\caption{Figure 1}
\label{fig:figure-1}
\end{figure}
\zcref{sec:section-1,fig:figure-1}
\end{document}

```

## 11 `\label` or `\zlabel`?

TL;DR: Just use `\label`, unless you have special requirements.

Technically, `zref`'s referencing system, and thus also `zref-clever`, require a label set with `\zlabel` to make a reference. However, the `labelhook` option (see Section 6) leverages the kernel's `label` hook to also set a `\zlabel` when a standard `\label` is called, so that we can simply use `\labels` in our document and refer to them with either referencing system. Indeed, in some places the use of `\label` this way may be required. That given, which is to be preferred: use `\label` all around or normally use `\zlabel` and, occasionally resort to `\label` where required? I guess it depends, but we can reason the pros and cons of both alternatives.

Simply using `\label` across your document clearly speaks for convenience. You don't have to worry with the exceptional case where a `\zlabel` may not work or setting it is not possible. You can use either referencing system for your labels as desired. Your favorite editor may have some facilities to ease the insertion of labels, but does not support `\zlabel`. And so on. The only disadvantage I can see with this approach is that two labels end up in the `.aux` file which, arguably, may be seen as a redundancy, or waste.

It is probably fair to consider this redundancy, in most use cases, as a negligible cost. But you may disagree, or the size of your document or your requirements may say otherwise, in which case you may prefer the second approach and use `\zlabel` normally, with some occasional `\label` where required. Outright disabling the `labelhook` option leaves you uncovered in such cases, but you can always locally (re-)enable it as needed. As far as my experience goes, the restrictions should not be that many, and some trial and error can easily tell what the situation is. See Sections 12 and 13 for some discussion and known cases.

All in all, as far as `zref-clever` is concerned, the use of `\label` throughout (with the `labelhook` option) is the recommended approach. Consider the use of `\zlabel` directly if you have some special requirements at stake and are willing to handle a few rough edges.

## 12 Limitations

Being based on `zref` entails one significant advantage for `zref-clever`: the extensible referencing system of the former allows `zref-clever` to store and retrieve the information it needs to work without having to redefine some core  $\LaTeX$  commands. Which leads to less compatibility problems and load order issues than some traditional cross-reference related packages. On the other hand, being based on `zref` also has some potential implications to the supported scope of `zref-clever`. Not because of any particular limitation of either, but because any class or package which implements some special handling for reference labels universally does so aiming at the standard referencing system, and whether specific support for `zref` is included, or whether things work by spillover of the particular technique employed, is not guaranteed.

The limitation here is less one of `zref-clever` than that of a potential lack of support for `zref` itself. Broadly speaking, what `zref-clever` does is setup `zref` so that its `\zref@newLabels` contains the information we need using `zref`'s API. Once the `\zlabel` is set correctly, there is little in the way of `zref-clever`, it can just extract the label's information, again using `zref`'s API, and do its job. Therefore, the problems that may arise are really in *label setting*.

For `\zlabel` to be able to set a label with everything `zref-clever` needs, some conditions must be fulfilled, most of which are pretty much the same as that of a regular label, but not only. As far as my experience goes, the following label setting requirements can be potentially problematic and are not necessarily granted for `\zlabel`:

1. One must be able to call `\zlabel`, directly or indirectly, at the appropriate scope/location so as to set the label.
2. When `\zlabel` is set, it must see a correct value of `\@currentcounter`.

As to the first, it is not everywhere we technically can set a (z)label. On verbatim-like environments it depends on how they are defined and whether they provide a proper place or option to do so. But other places may be problematic too, for example, `amsmath` display math environments also handle `\label` specially and the same work is not done for `\zlabel`. Classes and packages may offer label setting by means of arguments or options, and those usually only cater for the standard referencing system, etc. Fortunately for us, the `label` hook introduced by the  $\LaTeX$  kernel in the 2023-06-01 release improves the situation considerably. `zref-clever` makes use of it with the `labelhook` option, enabled by default, so that a standard `\label` also sets a `\zlabel` with the same name, thus providing a very general and reliable way to deal with this issue.

Regarding the second, a correctly set `\@currentcounter` is critical for the task of `zref-clever`: the reference type will depend on that and, consequently, sorting and compression as well, counter resetting behavior information is also retrieved based on it, and so on. Since the 2020-10-01  $\LaTeX$  release, `\@currentcounter` is set by `\refstepcounter` alongside `\@currentlabel` and, since the 2021-11-15 release, the support for `\@currentcounter` has been further extended in the kernel. Hence, as long as kernel features are involved, or as long as `\refstepcounter` is the tool used for the purpose of reference setting, the label will tend to have all information within its grasp at the time it is set. But that's not necessarily always the case. For this reason, `zref-clever` has the option `currentcounter` which at least allows for some viable work-arounds when the value of `\@currentcounter` cannot be relied upon. Whether we have a proper opening to set it, depends on the case. Still, `\refstepcounter` is ubiquitous enough a tool that we can count on `\@currentcounter` most of the time.

All in all, specially since the kernel's `label` hook has been made available, we can expect to find very little trouble in setting proper labels for `zref-clever`. Which is not to say "none", of course.

## 13 Compatibility modules

This section gives a description of each compatibility module provided by `zref-clever`. These modules intend to smooth the interaction of  $\LaTeX$  features, document classes, and packages with `zref-clever` and `zref`, and they can be selectively or completely disabled with the option `nocompat` (see Section 6). This set is not to be confused with "the list of packages or classes supported by `zref-clever`". In most circumstances, things should just

work out of the box, and need no specific handling. These are just the ones for which some special treatment was required.

Note that settings performed by the compatibility modules require special care if they are to be changed by users. Since the modules depend on what has been loaded or not in the document, the settings are done at `begindocument`. Hence, to override a setting performed by a compatibility module, the user must ensure their change comes after that, e.g. by adding their setting to the `begindocument` hook.

The purpose of outlining to some extent what the compatibility modules do is twofold. First, some of them may require usage adjustments, or awareness, which must be somehow conveyed in this documentation. Second, the kind and degree of intervention in external code varies for each module, and since this is an area of potential friction, a minimum of information for the users to judge whether they want to leave these modules enabled or not is due. For this reason, this is also a little technical, but for more details, see the code documentation.

`appendix`      The `\appendix` command provided by many document classes is normally used to change the behavior of the sectioning commands after that point. Usually, depending on the class, the changes that interest us involve using `\@Alph` for numbering and `\appendixname` for chapter's names. In sum, we'd like to refer to the appendix sectioning commands as "appendices" rather than "chapters" or "sections". Since the sectioning commands are the same as before `\appendix`, and so are their underlying counters, we must configure the counter type of the sectioning counters to `appendix`. Also, the sorting of appendix labels relative to those in the rest of the document requires attention, which is done by setting the main sectioning level counter as being reset by an internal counter with the `counterresetby` option. And this is what this compatibility module does, using a `ltxcmds` hook on `\appendix` for the purpose. Hence, this module applies to any document class or package which provides that command.

`appendices`    This module implements support for the `appendices` and `subappendices` environments provided by the `appendix` package, and also by `memoir`. The task is the same as for the `appendix` module: set proper counter types for the sectioning counters and ensuring proper sorting of labels. This module employs environment hooks to `appendices` and `subappendices`.

`memoir`        This compatibility module provides support for some of `memoir`'s cross-referencing features. Namely, it: i) sets counter types for counters `subfigure`, `subtable`, `poemline` (used in the `verse` environment), `sidefootnote`, and `pagenote`; ii) configures resetting behavior (`counterresetby` option) for `subfigure` and `subtable` counters; iii) provides the `zref` property "subcaption" so that we can use `ref=subcaption` to emulate the functionality of `memoir`'s `\subcaptionref`.

`amsmath`      `amsmath`'s `display math` environments have their contents processed twice, once for measuring and the second does the final typesetting. Hence, `amsmath` needs to handle `\label` specially inside these environments, otherwise we'd have duplicate labels all around, and indeed it does redefine `\label` locally inside them. Alas, the same treatment is not granted to `\zlabel`. Therefore, you must use `\label` (not `\zlabel`) inside `amsmath`'s `display math` environments, but unless you disabled the `labelhook` option you need not worry about that, and can just go with `\label` as usual. The following environments are subject to this usage restriction: `equation`, `align`, `alignat`, `flalign`, `xalignat`, `gather`, `multline`, and their respective starred versions. Besides that, the module ensures proper `currentcounter` values are in place for the `display math` environments, for which it uses environment hooks. The module also provides a `subeq` property, for `display math`



	environments used inside the <code>subequations</code> environment, which can be used to refer to them directly with the <code>ref</code> option, or to build terse ranges with the <code>endrange</code> option.
<code>mathtools</code>	<code>mathtools</code> has a feature to show the numbers only for those equations actually referenced (with <code>\eqref</code> or <code>\refeq</code> ), which is enabled by the <code>showonlyrefs</code> option. This compatibility module adds support for this feature, such that equation references made with <code>\zcref</code> also get marked as “referenced” for <code>mathtools</code> , when the option is active, of course. The module uses <code>mathtools’ \noeqref</code> for the purpose, but does not need to redefine or hook into anything, everything is handled on <code>zref-clever’s</code> side.
<code>breqn</code>	This compatibility module only sets proper <code>currentcounter</code> values for the environments <code>dgroup</code> , <code>dmath</code> , <code>dseries</code> , and <code>darray</code> , and uses environment hooks for the purpose. Note that, in part, this is needed because <code>breqn</code> does not use <code>\refstepcounter</code> to increment the equation counters and, as a result, fails to set <code>\@currentcounter</code> . But, for the same reason, it also fails to set <code>hyperref</code> anchors for the equations (thus affecting the standard labels too). So, you may wish to use the work-around provided by Heiko Oberdiek at <a href="https://tex.stackexchange.com/a/241150">https://tex.stackexchange.com/a/241150</a> (or equivalent).
<code>listings</code>	Being <code>lstlisting</code> a verbatim environment, setting labels inside it requires special treatment. <code>zref-clever’s</code> <code>labelhook</code> option provides that a label given to the <code>label</code> works out of the box, so unless you disabled the <code>labelhook</code> option you need not worry about it. Setting labels for specific lines of the environment can also be done, subject to escaping as due. The module sets appropriate <code>countertype</code> , <code>counterresetby</code> and <code>currentcounter</code> values for the <code>listings’</code> counters: <code>lstlisting</code> and <code>lstnumber</code> .
<code>enumitem</code>	$\LaTeX$ ’s <code>enumerate</code> environment requires some special treatment from <code>zref-clever</code> , since its resetting behavior is not stored in the standard way, and the counters’ names, given they are numbered by level, do not map to the reference type naturally. This is done by default for the up to four levels of nested <code>enumerate</code> environments the kernel offers. <code>enumitem</code> , though, allows one to increase this maximum list depth for <code>enumerate</code> and, if this is done, setup for these deeper nesting levels have also to be taken care of, and that’s what this compatibility module does. All settings here are internal to <code>zref-clever</code> , no hooks or redefinitions are needed, we just check the existing pertinent counters at <code>begindocument</code> , and supply settings for them. Of course, this means that only <code>enumitem’s</code> settings done in the preamble will be visible to the module and provided for.
<code>subcaption</code>	This compatibility module sets appropriate <code>countertype</code> and <code>counterresetby</code> for the <code>subfigure</code> and <code>subtable</code> counters, and provides the <code>zref</code> property “ <code>subref</code> ” so that we can refer to, for example, <code>\zcref[ref=subref]{(label)}</code> to emulate the functionality of <code>subcaption’s</code> <code>\subref</code> . The later feature uses the <code>\caption@subtypehook</code> provided by <code>caption</code> to locally add the <code>subref</code> property to <code>zref’s</code> main property list.
<code>subfig</code>	This module just sets appropriate <code>countertype</code> and <code>counterresetby</code> for the <code>subfigure</code> and <code>subtable</code> counters.

## 14 Work-arounds



The compatibility modules presented in Section 13 should go a long way in ensuring the user has little to worry about in setting up `zref-clever` for some more traditional document classes and packages, but they cannot be expected to go all the way. Not only because this kind of support will never be exhaustive, but also since, sometimes, given the way certain features are implemented, there may not be a reasonable way to provide this support, from our side. But, still, most of the time, it is still “viable” to get there, if one really wishes to do so. So, this section keeps track of some known recipes, which I don’t think belong in `zref-clever` itself, but which you may choose to use. Note that this list is intended

to spare users from having to reinvent the wheel every time someone needs something of the sort, but from zref-clever’s perspective, their status is “not supported”.

## beamer

beamer does some really atypical things with regard to cross-references. To start with, it redefines `\label` to receive an optional `<(overlay specification)>` argument. Then, presumably to support overlays, it goes on and hijacks `hyperref`’s anchoring system, sets anchors (`\hypertargets`) to each `label` in the `.snm` file, while letting every standard label’s anchor in the `.aux` file default to `Doc-Start`. Of course, having rendered useless `hyperref`’s anchoring, it has to redefine `\ref` so that it uses its own `.snm` provided “label anchors” to make hyperlinks. In particular, from our perspective, there is no support at all for `zref` provided by beamer. Which is specially unfortunate since the above procedures also appear to break `cleveref`.<sup>10</sup>

Adding proper support for this is the business of beamer, `zref`, and/or `hyperref`. Likely the former’s really. But, taking advantage of `zref`’s flexibility, as a user, you can have a work-around in the meantime.

### Work-around 2: beamer

```

\documentclass{beamer}
\usepackage{zref-clever}
\makeatletter
\RenewDocumentCommand{\zlabel}{D<>{1} m }{%
  \ifx\label\ltx@gobble
  \else
    \zref@wrapper@babel{\zref@label<#1>}{#2}%
  \fi
}
\NewCommandCopy\beamer@old@zref@label\zref@label
\RenewDocumentCommand{\zref@label}{D<>{1} m }{%
  \alt<#1>{%
    \zref@ifpropundefined{anchor}{\zref@setcurrent{anchor}{#2}}%
    \beamer@old@zref@label{#2}%
    \beamer@nameslide{#2}%
  }{%
    \beamer@dummyslide%
  }%
}
\makeatother
\begin{document}
\begin{frame}
\begin{table}
\begin{tabular}{cc}
1 & 2 \\
3 & 4
\end{tabular}
\caption{Table 1}
\zlabel{tab:1}

```

<sup>10</sup>See, for example, <https://tex.stackexchange.com/q/266080>, <https://tex.stackexchange.com/q/668998>, and <https://github.com/josephwright/beamer/issues/750>. The workaround provided at <https://tex.stackexchange.com/a/266109> is not general enough since it breaks `cleveref`’s ability to receive a list of labels as argument.

```

\end{table}
\end{frame}
\begin{frame}
\begin{figure}
\rule{5cm}{5cm}
\caption{Figure 1}
\zlabel{fig:1}
\end{figure}
\end{frame}
\begin{frame}
\zcref{tab:1,fig:1}
\end{frame}
\end{document}

```

This work-around redefines `\zlabel` so that it takes an overlay specification argument, and provides that the work done by beamer for the standard `\label` is also done for it. And it works by setting the anchor to the *label* so as to be able to speak the “beamer-lingo” of anchors.

A couple of caveats though. First, there’s probably still some work to be done there in defining and setting reference types for beamer specific document objects, e.g. overlays. But it can be done by the standard user interface of `zref-clever`. Second, since beamer’s anchoring system does not provide for uniqueness of anchors as `hyperref` does, if you (need to) use `\label` to set both `\label` and `\zlabel`, relying on the `labelhook` option, this will result in duplicate anchors for labels set in them, with corresponding `hyperref` warnings of “destination with the same identifier has been already used, duplicate ignored”. The warning is actually harmless in this case, since both labels are set in the same place, and thus have identical anchors, it is nevertheless there.

## 15 Acknowledgments

`zref-clever` would not be possible without other people’s previous work and help.

Heiko Oberdiek’s `zref`, now maintained by the Oberdiek Package Support Group, is the underlying infrastructure of this package. The potential of its basic concept and the solid implementation were certainly among the reasons I’ve chosen to venture into these waters, to start with. And I believe they will remain one of the main assets of `zref-clever` as it matures.

The name of the package makes no secret that a major inspiration for the kind of “feel” I strove to achieve has been Toby Cubitt’s `cleveref`. Indeed, I have been a user of `cleveref` for several years, and a happy one at that. But the role `cleveref` played in the development of `zref-clever` extends beyond the visible influence in the design of user facing functionality. Some technical solutions and, specially, the handling of support for other packages were a valuable reference. Hence, the accumulated experience of `cleveref` allowed for `zref-clever` to start on a more solid foundation than would otherwise be the case.

The long term efforts of the L<sup>A</sup>T<sub>E</sub>X Project Team around `expl3` and `xparse` have also left their marks in this package. By implementing powerful tools and smoothing several regular programming tasks, they have certainly reduced my entry barrier to L<sup>A</sup>T<sub>E</sub>X programming and enabled me to develop this package with a significantly reduced effort. And, given the constraints of my abilities, the result is no doubt much better than it would be in their absence.

Besides these more general acknowledgments, a number of people have contributed to `zref-clever`, whether they are aware of it or not. Suggestions, ideas, solutions to problems, bug reports or even encouragement were generously provided by (in chronological order): Ulrike Fischer, Phelype Oleinik, Enrico Gregorio, Steven B. Segletes, Jonathan P. Spratte, David Carlisle, Frank Mittelbach, ‘samcarter’, Alan Munn, Florent Rougon, Denis Bitouzé, Marcel Krüger, Jürgen Spitzmüller, ‘niluxv’, Joseph Wright, Thomas F. Sturm, Yukai Chou, and Lars Madsen.

The package’s language files have been provided or improved thanks to: Denis Bitouzé (French), François Lagarde (French), ‘niluxv’ (Dutch), Matteo Ferrigato (Italian), and Sergey Slyusarev (Russian).

If I have inadvertently left anyone off the list I apologize, and please let me know, so that I can correct the oversight.

Thank you all very much!

## 16 Change history

A change log with relevant changes for each version, eventual upgrade instructions, and upcoming changes, is maintained in the package’s repository, at <https://github.com/gusbrs/zref-clever/blob/main/CHANGELOG.md>. The change log is also distributed with the package’s documentation through CTAN upon release so, most likely, `texdoc zref-clever/changeLog` should provide easy local access to it. An archive of historical versions of the package is also kept at <https://github.com/gusbrs/zref-clever/releases>.