

# The zref-clever package<sup>\*</sup>

Gustavo Barros<sup>†</sup>

2021-09-29

## Abstract

`zref-clever` provides an user interface for making  $\text{\LaTeX}$  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 `\zceref` 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.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Loading the package</b>	<b>2</b>
<b>3</b>	<b>Dependencies</b>	<b>2</b>
<b>4</b>	<b>User interface</b>	<b>2</b>
<b>5</b>	<b>Options</b>	<b>2</b>
<b>6</b>	<b>Reference types</b>	<b>6</b>
<b>7</b>	<b>Reference format</b>	<b>7</b>
<b>8</b>	<b>Internationalization</b>	<b>7</b>
<b>9</b>	<b>Usage examples</b>	<b>7</b>
<b>10</b>	<b>Limitations</b>	<b>7</b>
<b>11</b>	<b>Acknowledgments</b>	<b>7</b>
<b>12</b>	<b>Change history</b>	<b>7</b>

---

<sup>\*</sup>This file describes v0.1.0-alpha, released 2021-09-29.

<sup>†</sup><https://github.com/gusbrs/zref-clever>

## 1 Introduction

## 2 Loading the package

As usual:

```
\usepackage[⟨options⟩]{zref-clever}
```

## 3 Dependencies

`zref-clever` requires `zref`, and L<sup>A</sup>T<sub>E</sub>X kernel 2021-06-01, or newer. It also needs `l3keys2e`. Some packages are leveraged by `zref-clever` if they are present, but are not loaded or required by it, namely: `hyperref`, `zref-titleref` (`zref`’s module), and `zref-check`.

## 4 User interface

---

<code>\zcref</code>	<code>\zcref⟨*⟩[⟨options⟩]{⟨labels⟩}</code>
---------------------	---

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  $\langle options \rangle$  are (mostly) the same as those of the package, and can be given to local effect.

---

<code>\zcpageref</code>	<code>\zcpageref⟨*⟩[⟨options⟩]{⟨labels⟩}</code>
-------------------------	---

Typesets page references to  $\langle labels \rangle$ , given as a comma separated list. It is equivalent to calling `\zcref` with the `ref=page` option: `\zcref⟨*⟩[⟨options⟩, ref=page]{⟨labels⟩}`.

---

<code>\zcsetup</code>	<code>\zcsetup{⟨options⟩}</code>
-----------------------	----------------------------------

Sets `zref-clever`’s general options (see Section 5).

---

<code>\zcRefTypeSetup</code>	<code>\zcRefTypeSetup {⟨type⟩} {⟨options⟩}</code>
------------------------------	---

Sets type-specific reference format options (see Section 7).

Besides these, user facing commands related to Internationalization are presented in Section 8.

## 5 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 7). General options can be set globally either as package options at load-time (see Section 2) or by means of `\zcsetup` in the preamble (see Section 4). They can also be set locally with `\zcsetup` along the document or through the optional argument of `\zcref` (see Section 4). 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.

<code>ref</code>	The <code>ref</code> option controls the label property to which <code>\zcref</code> refers to. It can receive values <code>zc@thecnt</code> and <code>page</code> . If <code>zref-titleref</code> is loaded, <code>ref</code> also accepts the value <code>title</code> . The package’s default is <code>zc@thecnt</code> , which is an internal property equivalent to <code>zref’s default</code> property, except that it is not affected by the kernel’s <code>\labelformat</code> . In sum, just what you’d expect from a regular reference. By default, sorting and compression is done according to the information of the counter underlying <code>zc@thecnt</code> . Special treatment in these areas is provided for <code>page</code> , but not for <code>title</code> . The <code>page</code> option is a convenience alias for <code>ref=page</code> .
<code>typeset</code> <code>noname</code>	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> option is a convenience alias for <code>typeset=ref</code> .
<code>sort</code> <code>nosort</code>	The <code>sort</code> option controls whether the list of <i>labels</i> received as argument by <code>\zcref</code> should be sorted or not. It is a boolean option, and defaults to <code>true</code> . The <code>nosort</code> option is a convenience alias for <code>sort=false</code> .
<code>typesort</code> <code>notypesort</code>	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, braced) can be placed anywhere along the list, to specify the sort priority of any type not included explicitly in the list. 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.
<code>comp</code> <code>nocomp</code>	<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, and defaults to <code>true</code> . The <code>nocomp</code> option is a convenience alias for <code>comp=false</code> . Of course, for better compression results the <code>sort</code> is recommended, but the two options are technically independent.
<code>range</code>	By default (that is, when the <code>range</code> option is not given), <code>\zcref</code> typesets a complete list of references according to the $\langle labels \rangle$ it received as argument, and only compresses some of them into ranges if the <code>comp</code> option is enabled and if references of the same type occur in immediate sequence. The <code>range</code> option makes <code>\zcref</code> behave differently. Sorting is implied by this option (the <code>sort</code> option is disregarded) and, for each reference type group in $\langle labels \rangle$ , <code>\zcref</code> builds a range from the first to the last reference in it, even if references in between do not occur in immediate sequence. <code>\zcref</code> is smart enough, though, 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”. It is a boolean option, and the package's default is <code>range=false</code> . The option given without a value is equivalent to <code>range=true</code> (in the l3keys' jargon, the <i>option's</i> default is <code>true</code> ).
<code>cap</code> <code>nocap</code> <code>capfirst</code>	The <code>cap</code> option controls whether the reference type names should be capitalized or not. It is a boolean option, and the package's default is <code>cap=false</code> . The option given without a value is equivalent to <code>cap=true</code> . The <code>nocap</code> option is a convenience alias for <code>cap=false</code> . The <code>capfirst</code> ensures that the reference type name of the <i>first</i> type block is capitalized, even when <code>cap</code> is set to <code>false</code> .
<code>abbrev</code> <code>noabbrev</code> <code>noabbrevfirst</code>	The <code>abbrev</code> option controls whether to use abbreviated reference type names when they are available. It is a boolean option, and the package's default is <code>abbrev=false</code> . The option given without a value is equivalent to <code>abbrev=true</code> . The <code>noabbrev</code> option is a convenience alias for <code>abbrev=false</code> . The <code>noabbrevfirst</code> ensures that the reference type name of the <i>first</i> type block is never abbreviated, even when <code>abbrev</code> is set to <code>true</code> .
<code>S</code>	<code>S</code> for “Sentence”. The <code>S</code> option is a convenience alias for <code>capfirst=true</code> , <code>noabbrevfirst=true</code> , 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 <code>S</code> option for beginning of sentence references. Even if you do happen to be currently using <code>cap=true</code> , <code>abbrev=false</code> , 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.
<code>hyperref</code>	The <code>hyperref</code> option controls the use of <code>hyperref</code> by <code>zref-clever</code> and takes values <code>auto</code> , <code>true</code> , <code>false</code> . The default value, <code>auto</code> , makes <code>zref-clever</code> use <code>hyperref</code> if it is loaded, meaning that references made with <code>\zcref</code> get hyperlinked to the anchors of their respective $\langle labels \rangle$ . <code>true</code> does the same thing, but warns if <code>hyperref</code> is not loaded ( <code>hyperref</code> is never loaded for you). In either of these cases, if <code>hyperref</code> is loaded, module <code>zref-hyperref</code> is also loaded by <code>zref-clever</code> . <code>false</code> means not to use <code>hyperref</code> regardless of its availability. This is a preamble only option, but <code>\zcref</code> provides granular control of hyperlinking by means of its starred version.
<code>nameinlink</code>	The <code>nameinlink</code> 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 <i>first</i> reference of each type block. <code>nameinlink</code> can receive values <code>true</code> , <code>false</code> , <code>single</code> , and <code>tsingle</code> . When the value is <code>true</code> the type name is always included in the hyperlink. When it is <code>false</code> the type name is never included in the link. When the value is <code>single</code> , the type name is included in the link only if <code>\zcref</code> 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 package's default 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 7). The default value, `main`, uses the main document language, as defined by `babel` or `polyglossia` (or `english` if none of them is loaded). Value `current` uses the current 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 8.

**font** The `font` option can receive font styling commands to change the appearance of the whole reference list (see also the reference format options, `namefont`, `reffont`, and `reffont-in` in Section 7). 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. Given how package options are handled by L<sup>A</sup>T<sub>E</sub>X, the fact that this option receives commands as value means this option *can't* be set at load time, as a package option. If you want to set it globally, use `\zcsetup` instead.

**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 7).

**check** Provides integration of `zref-clever` with the `zref-check` package. The `check` option is only available when the latter is loaded and, if so, it 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 whole reference list, including the `note`.

**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 6). A value like `countertype = {foo = bar}` sets the `foo` counter to use the reference type `bar`. There's only need to specify the `countertype` 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 default 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. Since this option only affects how labels are set, it is not available in `\zcref`.



**counterresetters**  
**counterresetby**

The sorting and compression of references done by `\zcref` requires that we know the counter whose `\refstepcounter` is being stored by `\zlabel` 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 adds counter names, received as a comma separated list, to the list of counters `zref-clever` uses to search for “enclosing counters” of the counter for which a label is being set. Unfortunately, not every counter gets reset through the standard machinery for this, including some L<sup>A</sup>T<sub>E</sub>X 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 default 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 default 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`.

## 6 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 L<sup>A</sup>T<sub>E</sub>X *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 `\zcref`. 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.

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. 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*.

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 “string” which `zref-clever` uses to look for a whole set of type-specific reference format options (see Section 7). 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.

## 7 Reference format

## 8 Internationalization

---

---

**`\zcDeclareLanguage`**

`\zcDeclareLanguage`  $\{\langle language \rangle\}$

Declare a new language for use with `zref-clever`. If  $\langle language \rangle$  is already known, just warn. `\zcDeclareLanguage` is preamble only.

---

---

**`\zcDeclareLanguageAlias`**

`\zcDeclareLanguageAlias`  $\{\langle language\ alias \rangle\} \{\langle aliased\ language \rangle\}$

Declare  $\langle language\ alias \rangle$  to be an alias of  $\langle aliased\ language \rangle$ .  $\langle aliased\ language \rangle$  must be already known to `zref-clever`. `\zcDeclareLanguageAlias` is preamble only.

---

---

**`\zcDeclareTranslations`**

`\zcDeclareTranslations`  $\{\langle language \rangle\} \{\langle options \rangle\}$

Sets language-specific reference format options for  $\langle language \rangle$  (see Section 7).  $\langle language \rangle$  must be already known to `zref-clever`. `\zcDeclareTranslations` is preamble only.

## 9 Usage examples

## 10 Limitations

## 11 Acknowledgments

## 12 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>.