

exp_kv

an expandable $\langle key \rangle = \langle value \rangle$ implementation

Jonathan P. Spratte*

2021-07-15 v1.9

Abstract

exp_kv provides a small interface for $\langle key \rangle = \langle value \rangle$ parsing. The parsing macro is *fully expandable*, the $\langle code \rangle$ of your keys might be not. exp_kv is *swift*, close to the fastest $\langle key \rangle = \langle value \rangle$ implementation. However it is the fastest which copes with active commas and equal signs and doesn't strip braces accidentally.

Contents

1	Documentation	2
1.1	Setting up Keys	2
1.2	Parsing Keys	4
1.3	Other Macros	6
1.4	Examples	10
1.4.1	Standard Use-Case	10
1.4.2	A Macro to Draw Rules	11
1.4.3	An Expandable $\langle key \rangle = \langle value \rangle$ Macro Using <code>\ekvsneak</code>	12
1.5	Error Messages	13
1.5.1	Load Time	14
1.5.2	Defining Keys	14
1.5.3	Using Keys	14
1.6	Bugs	15
1.7	Comparisons	15
1.8	License	17
2	Implementation	19
2.1	The L ^A T _E X Package	19
2.2	The Generic Code	19
	Index	41

*jspratte@yahoo.de

1 Documentation

`expkv` provides an expandable $\langle key \rangle = \langle value \rangle$ parser. The $\langle key \rangle = \langle value \rangle$ pairs should be given as a comma separated list and the separator between a $\langle key \rangle$ and the associated $\langle value \rangle$ should be an equal sign. Both, the commas and the equal signs, might be of category 12 (other) or 13 (active). To support this is necessary as for example `babel` turns characters active for some languages, for instance the equal sign is turned active for Turkish.

`expkv` is usable as generic code or as a \LaTeX package. To use it, just use one of:

```
\usepackage{expkv} % LaTeX
\input expkv      % plainTeX
```

The \LaTeX package doesn't do more than `expkv.tex`, except calling `\ProvidesPackage` and setting things up such that `expkv.tex` will use `\ProvidesFile`.

In the `expkv` family are other packages contained which provide additional functionality. Those packages currently are:

`expkvDEF` a key-defining frontend for `expkv` using a $\langle key \rangle = \langle value \rangle$ syntax

`expkvICS` define expandable $\langle key \rangle = \langle value \rangle$ macros using `expkv`

`expkvOPT` parse package and class options with `expkv`

Note that while the package names are stylised with a vertical rule, their names are all lower case with a hyphen (e.g., `expkv-def`).

A list of concise comparisons to other $\langle key \rangle = \langle value \rangle$ packages is contained in [subsection 1.7](#).

1.1 Setting up Keys

`expkv` provides a rather simple approach to setting up keys, similar to `keyval`. However there is an auxiliary package named `expkvDEF` which provides a more sophisticated interface, similar to well established packages like `pgfkeys` or `l3keys`.

Keys in `expkv` (as in almost all other $\langle key \rangle = \langle value \rangle$ implementations) belong to a *set* such that different sets can contain keys of the same name. Unlike many other implementations `expkv` doesn't provide means to set a default value, instead we have keys that take values and keys that don't (the latter are called `NoVal` keys by `expkv`), but both can have the same name (on the user level).

The following macros are available to define new keys. Those macros containing "def" in their name can be prefixed by anything allowed to prefix `\def` (but *don't* use `\outer`, keys defined with it won't ever be usable), prefixes allowed for `\let` can prefix those with "let" in their name, accordingly. Neither $\langle set \rangle$ nor $\langle key \rangle$ are allowed to be empty for new keys. $\langle set \rangle$ will be used as is inside of `\csname ... \endcsname` and $\langle key \rangle$ will get `\detokenized`.

`\ekvdef` `\ekvdef{\set}{\key}{\code}`

Defines a $\langle key \rangle$ taking a value in a $\langle set \rangle$ to expand to $\langle code \rangle$. In $\langle code \rangle$ you can use `#1` to refer to the given value.

Example: Define text in `foo` to store the value inside `\foo@text`:
`\protected\long\ekvdef{foo}{\text}{\def\foo@text{#1}}`

`\ekvdefNoVal``\ekvdefNoVal{<set>}{<key>}{<code>}`

Defines a no value taking `<key>` in a `<set>` to expand to `<code>`.

Example: Define `bool` in `foo` to set `\iffoo@bool` to `true`:

```
\protected\ekvdefNoVal{foo}{bool}{\foo@booltrue}
```

`\ekvlet``\ekvlet{<set>}{<key>}{<cs>}`

Let the value taking `<key>` in `<set>` to `<cs>`, there are no checks on `<cs>` enforced, but the code should expect the value as a single braced argument directly following it.

Example: Let `cmd` in `foo` do the same as `\foo@cmd`:

```
\ekvlet{foo}{cmd}\foo@cmd
```

`\ekvletNoVal``\ekvletNoVal{<set>}{<key>}{<cs>}`

Let the no value taking `<key>` in `<set>` to `<cs>`, it is not checked whether `<cs>` exists or that it takes no parameter.

Example: See above.

`\ekvletkv``\ekvletkv{<set>}{<key>}{<set2>}{<key2>}`

Let the `<key>` in `<set>` to `<key2>` in `<set2>`, it is not checked whether that second key exists (but take a look at `\ekvifdefined`).

Example: Let `B` in `bar` be an alias for `A` in `foo`:

```
\ekvletkv{bar}{B}{foo}{A}
```

`\ekvletkvNoVal``\ekvletkvNoVal{<set>}{<key>}{<set2>}{<key2>}`

Let the `<key>` in `<set>` to `<key2>` in `<set2>`, it is not checked whether that second key exists (but take a look at `\ekvifdefinedNoVal`).

Example: See above.

`\ekvdefunknown``\ekvdefunknown{<set>}{<code>}`

By default an error will be thrown if an unknown `<key>` is encountered. With this macro you can define `<code>` that will be executed for a given `<set>` when an unknown `<key>` with a `<value>` was encountered instead of throwing an error. You can refer to the given `<value>` with `#1` and to the unknown `<key>`'s name with `#2` in `<code>`.¹ `\ekvdefunknown` and `\ekvredirectunknown` are mutually exclusive, you can't use both.

Example: Also search `bar` for undefined keys of set `foo`:

```
\long\ekvdefunknown{foo}{\ekvset{bar}{#2=#1}}
```

This example differs from using `\ekvredirectunknown{foo}{bar}` (see below) in that also the unknown-key handler of the `bar` set will be triggered, error messages for undefined keys will look different, and this is slower than using `\ekvredirectunknown`.

`\ekvdefunknownNoVal``\ekvdefunknownNoVal{<set>}{<code>}`

As already explained for `\ekvdefunknown`, `expkv` would throw an error when encountering an unknown `<key>`. With this you can instead let it execute `<code>` if an unknown `NoVal <key>` was encountered. You can refer to the given `<key>` with `#1` in `<code>`. `\ekvdefunknownNoVal` and `\ekvredirectunknownNoVal` are mutually exclusive, you can't use both.

¹That order is correct, this way the code is faster.

Example: Also search bar for undefined keys of set foo:

```
\ekvdefunknownNoVal{foo}{\ekvset{bar}{#1}}
```

`\ekvredirectunknown` `\ekvredirectunknown{<set>}{<set-list>}`

This is a short cut to set up a special `\ekvdefunknown` for `<set>` that will check each set in the comma separated `<set-list>` for the unknown `<key>`. You can't use prefixes (so no `\long` or `\protected`) with this macro, the resulting unknown-key handler will always be `\long`. The first set in the `<set-list>` has highest priority. Once the `<key>` is found the remaining sets are discarded, if the `<key>` isn't found in any set an error will be thrown eventually. Note that the error messages are affected by the use of this macro, in particular, it isn't checked whether a `NoVal` key of the same name is defined in order to throw an unwanted value error. `\ekvdefunknown` and `\ekvredirectunknown` are mutually exclusive, you can't use both.

Example: For every key not defined in the set foo also search the sets bar and baz:

```
\ekvredirectunknown{foo}{bar, baz}
```

`\ekvredirectunknownNoVal` `\ekvredirectunknownNoVal{<set>}{<set-list>}`

This behaves just like `\ekvredirectunknown` and does the same but for the `NoVal` keys. Again no prefixes are supported. Note that the error messages are affected by the use of this macro, in particular, it isn't checked whether a normal key of the same name is defined in order to throw a missing value error. `\ekvdefunknownNoVal` and `\ekvredirectunknownNoVal` are mutually exclusive, you can't use both.

Example: See above.

1.2 Parsing Keys

`\ekvset` `\ekvset{<set>}{<key>=<value>, ...}`

Splits `<key>=<value>` pairs on commas. From both `<key>` and `<value>` up to one space is stripped from both ends, if then only a braced group remains the braces are stripped as well. So `\ekvset{foo}{bar=baz}` and `\ekvset{foo}{ {bar}= {baz} }` will both do `\<foobrcode>{baz}`, so you can hide commas, equal signs and spaces at the ends of either `<key>` or `<value>` by putting braces around them. If you omit the equal sign the code of the key created with the `NoVal` variants described in [subsection 1.1](#) will be executed. If `<key>=<value>` contains more than a single unhidden equal sign, it will be split at the first one and the others are considered part of the value. `\ekvset` should be nestable.

`\ekvset` is currently *not* alignment safe.² As a result, key names and values that contain an `&` must be wrapped in braces when `\ekvset` is used inside an alignment (like $\LaTeX 2_{\epsilon}$'s `tabular` environment) or you have to create a wrapper that ensures an alignment safe context.

Example: Parse `key=arg`, `key` in the set foo:

```
\ekvset{foo}{key=arg, key}
```

²This might change in the future, I've not decided yet.

`\ekvsetSneaked` `\ekvsetSneaked{<set>}{<sneak>}{<key>=<value>,...}`

Just like `\ekvset`, this macro parses the `<key>=<value>` pairs within the given `<set>`. But `\ekvsetSneaked` will behave as if `\ekvsneak` has been called with `<sneak>` as its argument as the first action.

Example: Parse `key=arg`, `key` in the set `foo` with `\afterwards` sneaked out:

```
\ekvsetSneaked{foo}{\afterwards}{key=arg, key}
```

`\ekvsetdef` `\ekvsetdef<cs>{<set>}`

With this function you can define a shorthand macro `<cs>` to parse keys of a specified `<set>`. It is always defined `\long`, but if you need to you can also prefix it with `\global`. The resulting macro is faster than but else equivalent to the idiomatic definition:

```
\long\def<cs>#1{\ekvset{<set>}{#1}}
```

Example: Define the macro `\foosetup` to parse keys in the set `foo` and use it to parse `key=arg`, `key`:

```
\ekvsetdef\foosetup{foo}  
\foosetup{key=arg, key}
```

`\ekvsetSneakeddef` `\ekvsetSneakeddef<cs>{<set>}`

Just like `\ekvsetdef` this defines a shorthand macro `<cs>`, but this macro will make it a shorthand for `\ekvsetSneaked`, meaning that `<cs>` will take two arguments, the first being stuff that should be given to `\ekvsneak` and the second the `<key>=<value>` list. The resulting macro is faster than but else equivalent to the idiomatic definition:

```
\long\def<cs>#1#2{\ekvsetSneaked{<set>}{#1}{#2}}
```

Example: Define the macro `\foothings` to parse keys in the set `foo` and accept a sneaked argument, then use it to parse `key=arg`, `key` and sneaked `\afterwards`:

```
\ekvsetSneakeddef\foothings{foo}  
\foothings{\afterwards}{key=arg, key}
```

`\ekvsetdefSneaked` `\ekvsetdefSneaked<cs>{<set>}{<sneaked>}`

And this one behaves like `\ekvsetSneakeddef` but with a fixed `<sneaked>` argument. So the resulting macro is faster than but else equivalent to the idiomatic definition:

```
\long\def<cs>#1{\ekvsetSneaked{<set>}{<sneaked>}{#1}}
```

Example: Define the macro `\barthing` to parse keys in the set `bar` and always execute `\afterwards` afterwards, then use it to parse `key=arg`, `key`:

```
\ekvsetdefSneaked\barthing{bar}{\afterwards}  
\barthing{key=arg, key}
```

`\ekvpars` `\ekvpars{<code1>}{<code2>}{<key>=<value>, ...}`

This macro parses the `<key>=<value>` pairs and provides those list elements which are only keys as an argument to `<code1>`, and those which are a `<key>=<value>` pair to `<code2>` as two arguments. It is fully expandable as well and returns each element of the parsed list in `\unexpanded`, which has no effect outside of an `\expanded` or `\edef` context. Also `\ekvpars` expands in exactly two steps of expansion. You can use multiple tokens in `<code1>` and `<code2>` or just a single control sequence name. In both cases the found `<key>` and `<value>` are provided as a brace group following them.

`\ekvpars` is alignment safe, meaning that you don't have to take any precautions if it is used inside an alignment context (like L^AT_EX 2_ε's `tabular` environment) and any key or value can contain an `&`.

`\ekvbreak`, `\ekvsneak`, and `\ekvchangeset` and their relatives don't work in `\ekvpars`. It is analogue to `expl3`'s `\keyval_parse:NNn`, but not with the same parsing rules – `\keyval_parse:NNn` throws an error on multiple equal signs per `<key>=<value>` pair and on empty `<key>` names in a `<key>=<value>` pair, both of which `\ekvpars` doesn't deal with.

Example:

```
\ekvpars{\handlekey{S}}{\handlekeyval{S}}{foo = bar, key, baz={zzz}}
```

would be equivalent to

```
\handlekeyval{S}{foo}{bar}\handlekey{S}{key}\handlekeyval{S}{baz}{zzz}
```

and afterwards `\handlekey` and `\handlekeyval` would have to further handle the `<key>`. There are no macros like these two contained in `expl3`, you have to set them up yourself if you want to use `\ekvpars` (of course the names might differ). If you need the results of `\ekvpars` as the argument for another macro, you should use `\expanded`, or expand `\ekvpars` twice, as only then the input stream will contain the output above:

```
\expandafter\parse\expanded{\b\ekvpars\k\kv{foo = bar, key, baz={zzz}}}
```

or

```
\expandafter\expandafter\expandafter  
\parse\b\ekvpars\k\kv{foo = bar, key, baz={zzz}}
```

would both expand to

```
\parse\kv{foo}{bar}\k{key}\kv{baz}{zzz}
```

1.3 Other Macros

`expl3` provides some other macros which might be of interest.

`\ekvVersion`
`\ekvDate`

These two macros store the version and date of the package.

`\ekvifdefined` `\ekvifdefined{<set>}{<key>}{<>true>}{<>false>}`
`\ekvifdefinedNoVal` `\ekvifdefinedNoVal{<set>}{<key>}{<>true>}{<>false>}`

These two macros test whether there is a `<key>` in `<set>`. It is false if either a hash table entry doesn't exist for that key or its meaning is `\relax`.

Example: Check whether the key `special` is already defined in set `foo`, if it isn't input a file that contains more key definitions:

```
\ekvifdefined{foo}{special}{}{\input{foo.morekeys.tex}}
```

```
\ekvifdefinedset \ekvifdefinedset{<set>}{<>true>}{<>false>}
```

This macro tests whether `<set>` is defined (which it is if at least one key was defined for it). If it is `<>true>` will be run, else `<>false>`.

Example: Check whether the set `VeRyUnLiKeLy` is already defined, if so throw an error, else do nothing:

```
\ekvifdefinedset{VeRyUnLiKeLy}
  {\errmessage{VeRyUnLiKeLy already defined}}{}
```

```
\ekvbreak \ekvbreak{<after>}
```

```
\ekvbreakPreSneak
\ekvbreakPostSneak
```

Gobbles the remainder of the current `\ekvset` macro and its argument list and reinserts `<after>`. So this can be used to break out of `\ekvset`. The first variant will also gobble anything that has been sneaked out using `\ekvsneak` or `\ekvsneakPre`, while `\ekvbreakPreSneak` will put `<after>` before anything that has been smuggled and `\ekvbreakPostSneak` will put `<after>` after the stuff that has been sneaked out.

Example: Define a key `abort` that will stop key parsing inside the set `foo` and execute `\foo@aborted`, or if it got a value `\foo@aborted@with`:

```
\ekvdefNoVal{foo}{abort}{\ekvbreak{\foo@aborted}}
\ekvdef{foo}{abort}{\ekvbreak{\foo@aborted@with{#1}}}
```

```
\ekvsneak \ekvsneak{<after>}
```

```
\ekvsneakPre
```

Puts `<after>` after the effects of `\ekvset`. The first variant will put `<after>` after any other tokens which might have been sneaked before, while `\ekvsneakPre` will put `<after>` before other smuggled stuff. This reads and reinserts the remainder of the current `\ekvset` macro and its argument list to do its job. After `\ekvset` has parsed the entire `<key>=<value>` list everything that has been `\ekvsneaked` will be left in the input stream. A small usage example is shown in [subsubsection 1.4.3](#).

Example: Define a key `secret` in the set `foo` that will sneak out `\foo@secretly@sneaked`:

```
\ekvdefNoVal{foo}{secret}{\ekvsneak{\foo@secretly@sneaked}}
```

```
\ekvchangeset \ekvchangeset{<new-set>}
```

Replaces the current set with `<new-set>`, so for the rest of the current `\ekvset` call, that call behaves as if it was called with `\ekvset{<new-set>}`. It is comparable to using `<key>/ .cd` in `pgfkeys`.

Example: Define a key `cd` in set `foo` that will change to another set as specified in the value, if the set is undefined it'll stop the parsing and throw an error as defined in the macro `\foo@cd@error`:

```
\ekvdef{foo}{cd}
  {\ekvifdefinedset{#1}{\ekvchangeset{#1}}{\ekvbreak{\foo@cd@error}}}
```

`\ekvoptarg` `\ekvoptarg{<next>}{<default>}`

This macro will check for a following optional argument in brackets (`[]`) expandably. After the optional argument there has to be a mandatory one. The code in `<next>` should expect two arguments (the processed optional argument and the mandatory one). If there was an optional argument the result will be `<next>{<optional>}<mandatory>` (so the optional argument will be wrapped in braces, the mandatory argument will be untouched). If there was no optional argument the result will be `<next>{<default>}<mandatory>` (so the default will be used and the mandatory argument will be wrapped in braces after being read once – if it was already wrapped it is effectively unchanged).

`\ekvoptarg` expands in exactly two steps, grabs all the arguments only at the second expansion step, and is alignment safe. It has its limitations however. It can't tell the difference between `[]` and `{[]}`, so it doesn't work if the mandatory argument is a single bracket. Also if the optional argument should contain a nested closing bracket, the optional argument has to use nested braces like so: `[{arg}ument]`.

Example: Say we have a macro that should take an optional argument defaulting to 1:

```
\newcommand\foo{\ekvoptarg\@foo{1}}
\newcommand\@foo[2]{Mandatory: #2\par Optional: #1}
```

`\ekvoptargTF` `\ekvoptargTF{<true>}{<false>}`

This macro is similar to `\ekvoptarg`, but will result in `<true>{<optional>}<mandatory>` or `<false>{<mandatory>}` instead of placing a default value.

`\ekvoptargTF` expands in exactly two steps, grabs all the arguments only at the second expansion step, and is alignment safe. It has the same limitations as `\ekvoptarg`.

Example: Say we have a macro that should behave differently depending on whether there was an optional argument or not. This could be done with:

```
\newcommand\foo{\ekvoptargTF\foo@a\foo@b}
\newcommand\foo@a[2]{Mandatory: #2\par Optional: #1}
\newcommand\foo@b[1]{Mandatory: #1\par No optional.}
```

`\ekvcsvloop` `\ekvcsvloop{<code>}{<csv-list>}`

This loops over the comma separated items in `<csv-list>` and, after stripping spaces from either end of `<item>` and removing at most one set of outer braces, leaves `\unexpanded{<code>{<item>}}` for each list item in the input stream. Blank elements are ignored (if you need a blank element it should be given as `{}`). It supports both active commas and commas of category other. You could consider it as a watered down version of `\ekvparse`. However it is not alignment safe, which you could achieve by nesting it in `\expanded` (since the braces around the argument of `\expanded` will hide `&s` from TeX's alignment parsing).

Example: The following splits a comma separated list and prints it in a typewriter font with parentheses around each element.

```
\newcommand* \myprocessor[1]{\texttt{(#1)}}
\ekvcsvloop \myprocessor{abc, def, ghi} \par
\ekvcsvloop \myprocessor{1,,2,,3,,4} \par
```

(abc) (def) (ghi)
(1) (2) (3) (4)

`\ekverr` `\ekverr{<package>}{<message>}`

This macro will throw an error fully expandably.³ The error length is limited to a total length of 69 characters, and since ten characters will be added for the formatting (`!` and `\Error:`) that leaves us with a total length for `<package>` plus `<message>` of 59 characters. If the message gets longer \TeX will only display the first 69 characters and append `\ETC.` to the end.

Neither `<package>` nor `<message>` expand any further. Also `<package>` must not contain an explicit `\par` token or the token `\thanks@jfbu`. No such restriction applies to `<message>`.

If `^^J` is set up as the `\newlinechar` (which is the case in $\LaTeX 2_{\epsilon}$ but not in plain \TeX by default) you can use that to introduce line breaks in your error message. However that doesn't change the message length limit.

After your own error message some further text will be placed. The formatting of that text will look good if `^^J` is the `\newlinechar`, else not so much. That text will read:

```
! Paragraph ended before \<an-expandable-macro>
completed due to above exception. If the error
summary is not comprehensible see the package
documentation.
I will try to recover now. If you're in inter-
active mode hit <return> at the ? prompt and I
continue hoping recovery was complete.
```

Any clean up has to be done by you, `\ekverr` will expand to nothing after throwing the error message.

Example: Say we set up a small calculation which works with user input. In our calculation we need a division, so have to watch out for division by zero. If we detect such a case we throw an error and do the recovery by using the biggest integer allowed in \TeX as the result.

```
\newcommand* \mydivision [2]
  {%
    \number \numexpr
      \ifnum \numexpr #2 = 0 % space here on purpose
        \ekverr {my} {division by 0. Setting result to 2147483647.}%
        2147483647%
      \else
        (#1) / (#2)%
      \fi
    \relax
  }
$(10+5)/(3-3)\approx\mydivision{10+5}{3-3}$
```

If that code gets executed the following will be the terminal output

```
Runaway argument?
! my Error: division by 0. Setting result to 2147483647.
! Paragraph ended before \<an-expandable-macro>
completed due to above exception. If the error
summary is not comprehensible see the package
```

³The used mechanism was to the best of my knowledge first implemented by Jean-François Burnol.

documentation.

I will try to recover now. If you're in interactive mode hit <return> at the ? prompt and I continue hoping recovery was complete.

<to be read again>

```
\par
1.15 $(10+5)/(3-3)\approx\mydivision{10+5}{3-3}
$
?
```

and the output would contain $(10 + 5)/(3 - 3) \approx 2147483647$ if we continued the T_EX run at the prompt.

<code>\ekv@name</code>	<code>\ekv@name{<set>}{<key>}</code>
<code>\ekv@name@set</code>	<code>\ekv@name@set{<set>}</code>
<code>\ekv@name@key</code>	<code>\ekv@name@key{<key>}</code>

The names of the macros that correspond to a key in a set are build with these macros. The name is built from two blocks, one that is formatting the <set> name (`\ekv@name@set`) and one for formatting the <key> name (`\ekv@name@key`). To get the actual name the argument to `\ekv@name@key` must be `\detokenized`. Both blocks are put together (with the necessary `\detokenize`) by `\ekv@name`. For NoVal keys an additional N gets appended irrespective of these macros' definition, so their name is `\ekv@name{<set>}{<key>}N`.

You can use these macros to implement additional functionality or access key macros outside of `explv`, but *don't* change them! `explv` relies on their exact definitions internally.

Example: Execute the callback of the NoVal key key in set foo:

```
\csname\ekv@name{foo}{key}N\endcsname
```

1.4 Examples

1.4.1 Standard Use-Case

Say we have a macro for which we want to create a <key>=<value> interface. The macro has a parameter, which is stored in the dimension `\ourdim` having a default value from its initialisation. Now we want to be able to change that dimension with the `width` key to some specified value. For that we'd do

```
\newdimen\ourdim
\ourdim=150pt
\protected\ekvdef{our}{width}{\ourdim=#1\relax}
```

as you can see, we use the set `our` here. We want the key to behave different if no value is specified. In that case the key should not use its initial value, but be smart and determine the available space from `\hsize`, so we also define

```
\protected\ekvdefNoVal{our}{width}{\ourdim=.9\hsize}
```

Now we set up our macro to use this <key>=<value> interface

```
\protected\def\ourmacro#1%
{\begingroup\ekvset{our}{#1}\the\ourdim\endgroup}
```

Finally we can use our macro like in the following

```
\ourmacro{} \par
\ourmacro{width} \par
\ourmacro{width=5pt} \par
```

```
150.0pt
145.08086pt
5.0pt
```

The same keys using `expkvDEF` Using `expkvDEF` we can set up the equivalent key using a `<key>=<value>` interface, after the following we could use `\ourmacro` in the same way as above. `expkvDEF` will allocate and initialise `\ourdim` and define the width key `\protected` for us, so the result will be exactly the same – with the exception that the default will use `\ourdim=.9\hsize\relax` instead.

```
\input expkv-def % or \usepackage{expkv-def}
\ekvdefinekeys{our}
{
  dimen width = \ourdim,
  qdefault width = .9\hsize,
  initial width = 150pt
}
```

1.4.2 A Macro to Draw Rules

Another small example could be a `<key>=<value>` driven `\rule` alternative, because I keep forgetting the correct order of its arguments. First we define the keys (and initialize the macros used to store the keys):

```
\makeatletter
\newcommand*\myrule@ht{1ex}
\newcommand*\myrule@wd{0.1em}
\newcommand*\myrule@raise{\z@}
\protected\ekvdef{myrule}{ht}{\def\myrule@ht{#1}}
\protected\ekvdef{myrule}{wd}{\def\myrule@wd{#1}}
\protected\ekvdef{myrule}{raise}{\def\myrule@raise{#1}}
\protected\ekvdef{myrule}{lower}{\def\myrule@raise{-#1}}
```

Then we define a macro to change the defaults outside of `\myrule` and `\myrule` itself:

```
\ekvsetdef\myruleset{myrule}
\newcommand*\myrule[1][ ]
{ \begingroup \myruleset{#1} \myrule@out \endgroup }
```

And finally the output:

```
\newcommand*\myrule@out{\rule[\myrule@raise]\myrule@wd\myrule@ht}
\makeatother
```

And we can use it:

```
a\myrule \par
a\myrule[ht=2ex, lower=.5ex] \par
\myruleset{wd=5pt}
a\myrule
```

```
a|
a|
a■
```

1.4.3 An Expandable $\langle key \rangle = \langle value \rangle$ Macro Using `\ekvsneak`

Let's set up an expandable macro, that uses a $\langle key \rangle = \langle value \rangle$ interface. The problems we'll face for this are:

1. ignoring duplicate keys
2. default values for keys which weren't used
3. providing the values as the correct argument to a macro (ordered)

First we need to decide which $\langle key \rangle = \langle value \rangle$ parsing macro we want to do this with, `\ekvset` or `\ekvparse`. For this example we also want to show the usage of `\ekvsneak`, hence we'll choose `\ekvset`. And we'll have to use `\ekvset` such that it builds a parsable list for our macro internals. To gain back control after `\ekvset` is done we have to put an internal of our macro at the start of that list, so we use an internal key that uses `\ekvsneakPre` after any user input.

To ignore duplicates will be easy if the value of the key used last will be put first in the list, so the following will use `\ekvsneakPre` for the user-level keys. If we wanted some key for which the first usage should be the binding one we would use `\ekvsneak` instead for that key.

Providing default values can be done in different ways, we'll use a simple approach in which we'll just put the outcome of our keys if they were used with default values before the parsing list terminator.

Ordering the keys can be done simply by searching for a specific token for each argument which acts like a flag, so our sneaked out values will include specific tokens acting as markers.

Now that we have answers for our technical problems, we have to decide what our example macro should do. How about we define a macro that calculates the sine of a number and rounds that to a specified precision? As a small extra this macro should understand input in radian and degree and the used trigonometric function should be selectable as well. For the hard part of this task (expandably evaluating trigonometric functions) we'll use the `xfp` package.

First we set up our keys according to our earlier considerations and set up the user facing macro `\sine`. The end marker of the parsing list will be a `\sine@stop` token, which we don't need to define and we put our defaults right before it. The user macro `\sine` uses `\ekvoptargTF` to check for the optional argument short cutting to the final step if no optional argument was found. This way we save some time in this case, though we have to specify the default values twice.

```
\RequirePackage{xfp}
\makeatletter
\ekvdef{expex}{f}{\ekvsneakPre{\f{#1}}}
\ekvdef{expex}{round}{\ekvsneakPre{\rnd{#1}}}
\ekvdefNoVal{expex}{degree}{\ekvsneakPre{\deg{d}}}
\ekvdefNoVal{expex}{radian}{\ekvsneakPre{\deg{}}}
\ekvdefNoVal{expex}{internal}{\ekvsneakPre{\sine@rnd}}
\newcommand*\sine{\ekvoptargTF\sine@args{\sine@final{sin}{d}{3}}
\newcommand*\sine@args[2]
{\ekvset{expex}{#1,internal}\rnd{3}\deg{d}\f{sin}\sine@stop{#2}}
```

Now we need to define some internal macros to extract the value of each key's last usage (remember that this will be the group after the first special flag-token). For that we use one delimited macro per key.

```
\def\sine@rnd#1\rnd#2#3\sine@stop{\sine@deg#1#3\sine@stop{#2}}
\def\sine@deg#1\deg#2#3\sine@stop{\sine@f#1#3\sine@stop{#2}}
\def\sine@f#1\f#2#3\sine@stop{\sine@final{#2}}
```

After the macros `\sine@rnd`, `\sine@deg`, and `\sine@f` the macro `\sine@final` will see `\sine@final{<f>}{<degree/radian>}{<round>}{<num>}`. Now `\sine@final` has to expandably deal with those arguments such that the `\fpeval` macro of `xfp` gets the correct input. Luckily this is pretty straight forward in this example. In `\fpeval` the trigonometric functions have names such as `sin` or `cos` and the degree taking variants `sind` or `cosd`. And since the `degree` key puts a `d` in `#2` and the `radian` key leaves `#2` empty all we have to do to get the correct function name is stick the two together.

```
\newcommand*\sine@final[4]{\fpeval{round(#1#2(#4),#3)}}
\makeatother
```

Let's test our macro:

```
\sine{60}\par
\sine[round=10]{60}\par
\sine[f=cos , radian]{pi}\par
\edef\myval{\sine[f=tan]{1}}\texttt{\meaning\myval}
```

```
0.866
0.8660254038
-1
macro:->0.017
```

The same macro using `expkvics` Using `expkvics` we can set up something equivalent with a bit less code. The implementation chosen in `expkvics` is more efficient than the example above and way easier to code for the user.

```
\makeatletter
\newcommand*\sine{\ekvoptargTF\sine@a{\sine@b{sin}{d}{3}}
\ekvcSplitAndForward\sine@a\sine@b
{
  f=sin ,
  unit=d ,
  round=3 ,
}
\ekvcSecondaryKeys\sine@a
{
  nmeta degree={unit=d} ,
  nmeta radian={unit={}} ,
}
\newcommand*\sine@b[4]{\fpeval{round(#1#2(#4),#3)}}
\makeatother
```

The resulting macro will behave just like the one previously defined, but will have an additional `unit` key, since in `expkvics` every argument must have a value taking key which defines it.

1.5 Error Messages

`expkv` should only send messages in case of errors, there are no warnings and no info messages. In this subsection those errors are listed.

1.5.1 Load Time

`expkv.tex` checks whether ϵ -TeX and the `\expanded` primitive are available. If it isn't, an error will be thrown using `\errmessage`:

```
! expkv Error: e-TeX and \expanded required.
```

1.5.2 Defining Keys

If you get any error from `expkv` while you're trying to define a key, the definition will be aborted and gobbled.

If you try to define a key with an empty set name you'll get:

```
! expkv Error: empty set name not allowed.
```

Similarly, if you try to define a key with an empty key name:

```
! expkv Error: empty key name not allowed.
```

Both of these messages are done in a way that doesn't throw additional errors due to `\global`, `\long`, etc., not being used correctly if you prefixed one of the defining macros.

1.5.3 Using Keys

This subsection contains the errors thrown during `\ekvset`. The errors are thrown in an expandable manner using `\ekvrr`. In the following messages `<key>` gets replaced with the problematic key's name, and `<set>` with the corresponding set. If any errors during `<key>=<value>` handling are encountered, the entry in the comma separated list will be omitted after the error is thrown and the next `<key>=<value>` pair will be parsed.

If you're using an undefined key you'll get:

```
Runaway argument?
```

```
! expkv Error: unknown key '<key>' in set '<set>'
```

If you're using a key for which only a normal version and no `NoVal` version is defined, but don't provide a value, you'll get:

```
Runaway argument?
```

```
! expkv Error: missing value for '<key>' in set '<set>'
```

If you're using a key for which only a `NoVal` version and no normal version is defined, but provide a value, you'll get:

```
Runaway argument?
```

```
! expkv Error: unwanted value for '<key>' in set '<set>'
```

If you're using an undefined key in a set for which `\ekvredirectunknown` was used, and the key isn't found in any of the other sets as well, you'll get:

```
Runaway argument?
```

```
! expkv Error: no key '<key>' in sets {<set1>}{<set2>}...
```

If you're using an undefined `NoVal` key in a set for which `\ekvredirectunknownNoVal` was used, and the key isn't found in any of the other sets as well, you'll get:

```
Runaway argument?
```

```
! expkv Error: no NoVal key '<key>' in sets {<set1>}{<set2>}...
```

If you're using a set for which you never executed one of the defining macros from [subsection 1.1](#) you'll get a low level T_EX error, as that isn't actively tested by the parser (and hence will lead to undefined behaviour and not be gracefully ignored). The error will look like

```
! Missing \endcsname inserted.
<to be read again>
      \! expkv Error: Set '\set' undefined.
```

1.6 Bugs

Just like `keyval`, `expkv` is bug free. But if you find [bugshidden features](#)⁴ you can tell me about them either via mail (see the first page) or directly on GitHub if you have an account there: https://github.com/Skillmon/tex_expkv

1.7 Comparisons

This subsection makes some basic comparison between `expkv` and other $\langle key \rangle = \langle value \rangle$ packages. The comparisons are really concise, regarding speed, feature range (without listing the features of each package), and bugs and misfeatures.

Comparisons of speed are done with a very simple test key and the help of the `l3benchmark` package. The key and its usage should be equivalent to

```
\protected\ekvdef{test}{height}{\def\myheight{#1}}
\ekvsetdef\expkvtest{test}
\expkvtest{ height = 6 }
```

and only the usage of the key, not its definition, is benchmarked. For the impatient, the essence of these comparisons regarding speed and buggy behaviour is contained in [Table 1](#).

As far as I know `expkv` is the only fully expandable $\langle key \rangle = \langle value \rangle$ parser. I tried to compare `expkv` to every $\langle key \rangle = \langle value \rangle$ package listed on [CTAN](#), however, one might notice that some of those are missing from this list. That's because I didn't get the others to work due to bugs, or because they just provide wrappers around other packages in this list.

In this subsection is no benchmark of `\ekvparse` and `\keyval_parse:NNn` contained, as most other packages don't provide equivalent features to my knowledge. `\ekvparse` is slightly faster than `\ekvset`, but keep in mind that it does less. The same is true for `\keyval_parse:NNn` compared to `\keys_set:nn` of `expl3` (where the difference is much bigger). Comparing just the two, `\ekvparse` is a tad faster than `\keyval_parse:NNn` because of the two tests (for empty key names and only a single equal sign) which are omitted.

`keyval` is about 30% to 40% faster and has a comparable feature set (actually a bit smaller since `expkv` supports unknown-key handlers and redirection to other sets) just a slightly different way how it handles keys without values. That might be considered a drawback, as it limits the versatility, but also as an advantage, as it might reduce doubled code. Keep in mind that as soon as someone loads `xkeyval` the performance of `keyval` gets replaced by `xkeyval`'s.

⁴Thanks, David!

Also `keyval` has a bug, which unfortunately can't really be resolved without breaking backwards compatibility for *many* documents, namely it strips braces from the argument before stripping spaces if the argument isn't surrounded by spaces, also it might strip more than one set of braces. Hence all of the following are equivalent in their outcome, though the last two lines should result in something different than the first two:

```
\setkeys{foo}{bar=bar}
\setkeys{foo}{bar= {bar}}
\setkeys{foo}{bar={ bar}} % should be ' bar'
\setkeys{foo}{bar={{bar}}} % should be '{{bar}}'
```

`xkeyval` is roughly twenty times slower, but it provides more functionality, e.g., it has choice keys, boolean keys, and so on. It contains the same bug as `keyval` as it has to be compatible with it by design (it replaces `keyval`'s frontend), but also adds even more cases in which braces are stripped that shouldn't be stripped, worsening the situation.

`ltxkeys` is no longer compatible with the L^AT_EX kernel starting with the release 2020-10-01. It is over 380 times slower – which is funny, because it aims to be “[...] faster [...] than these earlier packages [referring to `keyval` and `xkeyval`].” It needs more time to parse zero keys than five of the packages in this comparison need to parse 100 keys. Since it aims to have a bigger feature set than `xkeyval`, it most definitely also has a bigger feature set than `explv`. Also, it can't parse `\long` input, so as soon as your values contain a `\par`, it'll throw errors. Furthermore, `ltxkeys` doesn't strip outer braces at all by design, which, imho, is a weird design choice. In addition `ltxkeys` loads `catoptions` which is known to introduce bugs (e.g., see <https://tex.stackexchange.com/questions/461783>). Because it is no longer compatible with the kernel, I stop benchmarking it (so the numbers listed here and in [Table 1](#) regarding `ltxkeys` were last updated on 2020-10-05).

`l3keys` is around four and a half times slower, but has an, imho, great interface to define keys. It strips *all* outer spaces, even if somehow multiple spaces ended up on either end. It offers more features, but is pretty much bound to `expl3` code. Whether that's a drawback is up to you.

`pgfkeys` is around 2.7 times slower for one key if one uses the `//<path>/ .cd` syntax and almost 20% slower if one uses `\pgfqkeys`, but has an *enormous* feature set. To get the best performance `\pgfqkeys` was used in the benchmark. This reduces the overhead for setting the base directory of the benchmark keys by about 43 ops (so both p_0 and T_0 would be about 43 ops bigger if `\pgfkeys{<path>/ .cd, <keys>}` was used instead). It has the same or a very similar bug `keyval` has. The brace bug (and also the category fragility) can be fixed by `pgfkeyx`, but this package was last updated in 2012 and it slows down `\pgfkeys` by factor 8. Also `pgfkeyx` is no longer compatible with versions of `pgfkeys` newer than 2020-05-25.

`kvsetkeys` with `kvdefinekeys` is about 4.4 times slower, but it works even if commas and equals have category codes different from 12 (just as some other packages in this list). Else the features of the keys are equal to those of `keyval`, the parser has more features, though.

options is 1.7 times slower for only a single value. It has a much bigger feature set. Unfortunately it also suffers from the premature unbracing bug **keyval** has.

simplekv is hard to compare because I don't speak French (so I don't understand the documentation). There was an update released on 2020-04-27 which greatly improved the package's performance and adds functionality so that it can be used more like most of the other $\langle key \rangle = \langle value \rangle$ packages. It has problems with stripping braces and spaces in a hard to predict manner just like **keyval**. Also, while it tries to be robust against category code changes of commas and equal signs, the used mechanism fails if the $\langle key \rangle = \langle value \rangle$ list already got tokenised. Regarding unknown keys it got a very interesting behaviour. It doesn't throw an error, but stores the $\langle value \rangle$ in a new entry accessible with `\useKV`. Also if you omit $\langle value \rangle$ it stores `true` for that $\langle key \rangle$. For up to three keys, **explkv** is a bit faster, for more keys **simplekv** takes the lead.

YAX is over twenty times slower. It has a pretty strange syntax for the \TeX -world, imho, and again a direct equivalent is hard to define (don't understand me wrong, I don't say I don't like the syntax, it's just atypical). It has the premature unbracing bug, too. Also somehow loading **YAX** broke options for me. The tested definition was:

```
\usepackage{yax}
\defactiveparameter yax {\storevalue\myheight yax:height } % setup
\setparameterlist{yax}{ height = 6 } % benchmark
```

1.8 License

Copyright © 2020–2021 Jonathan P. Spratte

This work may be distributed and/or modified under the conditions of the \LaTeX Project Public License (LPPL), either version 1.3c of this license or (at your option) any later version. The latest version of this license is in the file:

<http://www.latex-project.org/lppl.txt>

This work is “maintained” (as per LPPL maintenance status) by
Jonathan P. Spratte.

Table 1: Comparison of $\langle key \rangle = \langle value \rangle$ packages. The packages are ordered from fastest to slowest for one $\langle key \rangle = \langle value \rangle$ pair. Benchmarking was done using `l3benchmark` and the scripts in the `Benchmarks` folder of the [git repository](#). The columns p_i are the polynomial coefficients of a linear fit to the run-time, p_0 can be interpreted as the overhead for initialisation and p_1 the cost per key. The T_0 column is the actual mean ops needed for an empty list argument, as the linear fit doesn't match that point well in general. The column "BB" lists whether the parsing is affected by some sort of brace bug, "CF" stands for category code fragile and lists whether the parsing breaks with active commas or equal signs.

Package	p_1	p_0	T_0	BB	CF	Date
keyval	13.7	1.5	7.3	yes	yes	2014-10-28
expkv	19.7	2.2	6.6	no	no	2020-10-10
simplekv	18.3	7.0	17.7	yes	yes	2020-04-27
pgfkeys	24.3	1.7	10.7	yes	yes	2020-09-05
options	23.6	15.6	20.8	yes	yes	2015-03-01
kvsetkeys	*	*	40.3	no	no	2019-12-15
l3keys	71.3	33.1	31.6	no	no	2020-09-24
xkeyval	253.6	202.2	168.3	yes	yes	2014-12-03
YAX	421.9	157.0	114.7	yes	yes	2010-01-22
ltxkeys	3400.1	4738.0	5368.0	no	no	2012-11-17

*For `kvsetkeys` the linear model used for the other packages is a poor fit, `kvsetkeys` seems to have approximately quadratic run-time, the coefficients of the second degree polynomial fit are $p_2 = 8.2$, $p_1 = 44.9$, and $p_0 = 60.8$. Of course the other packages might not really have linear run-time, but at least from 1 to 20 keys the fits don't seem too bad. If one extrapolates the fits for 100 $\langle key \rangle = \langle value \rangle$ pairs one finds that most of them match pretty well, the exception being `ltxkeys`, which behaves quadratic as well with $p_2 = 23.5$, $p_1 = 2906.6$, and $p_0 = 6547.5$.

2 Implementation

2.1 The L^AT_EX Package

First we set up the L^AT_EX package. That one doesn't really do much except `\inputting` the generic code and identifying itself as a package.

```
1 \def\ekv@tmp
2   {%
3     \ProvidesFile{expkv.tex}%
4     [\ekvDate\space v\ekvVersion\space an expandable key=val implementation]%
5   }
6 \input{expkv.tex}
7 \ProvidesPackage{expkv}%
8   [\ekvDate\space v\ekvVersion\space an expandable key=val implementation]
```

2.2 The Generic Code

The rest of this implementation will be the generic code.

We make sure that it's only input once:

```
9 \expandafter\ifx\csname ekvVersion\endcsname\relax
10 \else
11 \expandafter\endinput
12 \fi
    Check whether  $\varepsilon$ -TEX and \expanded are available – expkv requires  $\varepsilon$ -TEX.
13 \begingroup
14 \edef\ekvtmpa{\string\expanded}
15 \edef\ekvtmpb{\meaning\expanded}
16 \expandafter
17 \endgroup
18 \ifx\ekvtmpa\ekvtmpb
19 \else
20 \errmessage{expkv Error: e-TeX and \noexpand\expanded required}
21 \expandafter\endinput
22 \fi
```

`\ekvVersion` We're on our first input, so let's store the version and date in a macro.

```
\ekvDate
23 \def\ekvVersion{1.9}
24 \def\ekvDate{2021-07-15}
```

(End definition for `\ekvVersion` and `\ekvDate`. These functions are documented on page 6.)

If the L^AT_EX format is loaded we want to be a good file and report back who we are, for this the package will have defined `\ekv@tmp` to use `\ProvidesFile`, else this will expand to a `\relax` and do no harm.

```
25 \csname ekv@tmp\endcsname
```

Store the category code of `@` to later be able to reset it and change it to `11` for now.

```
26 \expandafter\chardef\csname ekv@tmp\endcsname=\catcode'\@
27 \catcode'\@=11
```

`\ekv@tmp` might later be reused to gobble any prefixes which might be provided to `\ekvdef` and similar in case the names are invalid, we just temporarily use it here as means to store the current category code of `@` to restore it at the end of the file, we never care for the actual definition of it.

`\ekv@if@lastnamedcs` If the primitive `\lastnamedcs` is available, we can be a bit faster than without it. So we test for this and save the test's result in this macro.

```

28 \begingroup
29   \edef\ekv@tmpa{\string \lastnamedcs}
30   \edef\ekv@tmpb{\meaning\lastnamedcs}
31   \ifx\ekv@tmpa\ekv@tmpb
32     \def\ekv@if@lastnamedcs{\long\def\ekv@if@lastnamedcs##1##2{##1}}
33   \else
34     \def\ekv@if@lastnamedcs{\long\def\ekv@if@lastnamedcs##1##2{##2}}
35   \fi
36   \expandafter
37 \endgroup
38 \ekv@if@lastnamedcs

```

(End definition for `\ekv@if@lastnamedcs`.)

`\ekv@empty` Sometimes we have to introduce a token to prevent accidental brace stripping. This token would then need to be removed by `\@gobble` or similar. Instead we can use `\ekv@empty` which will just expand to nothing, that is faster than gobbling an argument.

```

39 \def\ekv@empty{}

```

(End definition for `\ekv@empty`.)

`\@gobble` Since branching tests are often more versatile than `\if... \else... \fi` constructs, we define helpers that are branching pretty fast. Also here are some other utility functions that just grab some tokens. The ones that are also contained in L^AT_EX don't use the `ekv` prefix. Not all of the ones defined here are really needed by `explk` but are provided because packages like `explkDEF` or `explkOPT` need them (and I don't want to define them in each package which might need them).

```

40 \long\def\@gobble#1{}
41 \long\def\@firstofone#1{#1}
42 \long\def\@firstoftwo#1#2{#1}
43 \long\def\@secondoftwo#1#2{#2}
44 \long\def\ekv@fi@gobble\fi\@firstofone#1{\fi}
45 \long\def\ekv@fi@firstofone\fi\@gobble#1{\fi#1}
46 \long\def\ekv@fi@firstoftwo\fi\@secondoftwo#1#2{\fi#1}
47 \long\def\ekv@fi@secondoftwo\fi\@firstoftwo#1#2{\fi#2}
48 \def\ekv@gobble@mark\ekv@mark{}
49 \long\def\ekv@gobbleto@stop#1\ekv@stop{}
50 \long\def\ekv@gobble@from@mark@to@stop\ekv@mark#1\ekv@stop{}

```

(End definition for `\@gobble` and others.)

As you can see `\ekv@gobbleto@stop` uses a special marker `\ekv@stop`. The package will use three such markers, the one you've seen already, `\ekv@mark` and `\ekv@nil`. Contrarily to how for instance `expl3` does things, we don't define them, as we don't need them to have an actual meaning. This has the advantage that if they somehow get expanded – which should never happen if things work out – they'll throw an error directly.

`\ekv@ifempty` We can test for a lot of things building on an if-empty test, so let's define a really fast one. Since some tests might have reversed logic (true if something is not empty) we also set up macros for the reversed branches.

```

51 \long\def\ekv@ifempty#1%
\ekv@ifempty@
\ekv@ifempty@true
\ekv@ifempty@false
\ekv@ifempty@true@F
\ekv@ifempty@true@F@gobble
\ekv@ifempty@true@F@gobbletwo

```

```

52   {%
53     \ekv@ifempty@A\ekv@ifempty@B\ekv@ifempty@true
54     \ekv@ifempty@A\ekv@ifempty@B\@secondoftwo
55   }
56 \long\def\ekv@ifempty@#1\ekv@ifempty@A\ekv@ifempty@B{
57 \long\def\ekv@ifempty@true\ekv@ifempty@A\ekv@ifempty@B\@secondoftwo#1#2{#1}
58 \long\def\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo#1#2{#2}
59 \long\def\ekv@ifempty@true@F\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1{
60 \long\def\ekv@ifempty@true@F@gobble\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1#2%
61   {}
62 \long\def\ekv@ifempty@true@F@gobbletwo
63   \ekv@ifempty@A\ekv@ifempty@B\@firstofone#1#2#3%
64   {}

```

(End definition for \ekv@ifempty and others.)

`\ekv@ifblank` The obvious test that can be based on an if-empty is if-blank, meaning a test checking whether the argument is empty or consists only of spaces. Our version here will be tweaked a bit, as we want to check this, but with one leading `\ekv@mark` token that is to be ignored. The wrapper `\ekv@ifblank` will not be used by `explkv` for speed reasons but `explkv|OPT` uses it.

```

65 \long\def\ekv@ifblank#1%
66   {%
67     \ekv@ifblank@#1\ekv@nil\ekv@ifempty@B\ekv@ifempty@true
68     \ekv@ifempty@A\ekv@ifempty@B\@secondoftwo
69   }
70 \long\def\ekv@ifblank@\ekv@mark#1{\ekv@ifempty@\ekv@ifempty@A}

```

(End definition for \ekv@ifblank and \ekv@ifblank@.)

`\ekv@ifdefined` We'll need to check whether something is defined quite frequently, so why not define a macro that does this. The following test is expandable and pretty fast. The version with `\lastnamedcs` is the fastest version to test for an undefined macro I know of (that considers both undefined macros and those with the meaning `\relax`).

```

71 \ekv@if@lastnamedcs
72   {%
73     \def\ekv@ifdefined#1{\ifcsname#1\endcsname\ekv@ifdef@fi\@secondoftwo}
74     \def\ekv@ifdef@fi\@secondoftwo
75       {%
76         \fi
77         \expandafter\ifx\lastnamedcs\relax
78         \ekv@fi@secondoftwo
79         \fi
80         \@firstoftwo
81       }
82   }
83   {%
84     \def\ekv@ifdefined#1%
85       {%
86         \ifcsname#1\endcsname\ekv@ifdef@fi\ekv@ifdef@false#1\endcsname\relax
87         \ekv@fi@secondoftwo
88         \fi
89         \@firstoftwo
90       }

```

```

91 \def\ekv@ifdef@fi\ekv@ifdef@false{\fi\expandafter\ifx\cename}
92 \long\def\ekv@ifdef@false
93   #1\endcsname\relax\ekv@fi@secondoftwo\fi\@firstoftwo#2#3%
94   {#3}
95 }

```

(End definition for `\ekv@ifdefined`.)

`\ekv@strip` We borrow some ideas of `expl3`'s `l3tl` to strip spaces from keys and values. This `\ekv@strip` also strips one level of outer braces *after* stripping spaces, so an input of `{abc}` becomes `abc` after stripping. It should be used with `#1` prefixed by `\ekv@mark`. Also this implementation at most strips *one* space from both sides (which should be fine most of the time, since `TEX` reads consecutive spaces as a single one during tokenisation).

```

96 \def\ekv@strip#1%
97   {%
98   \long\def\ekv@strip##1%
99     {%
100     \ekv@strip@a
101     ##1\ekv@nil
102     \ekv@mark#1%
103     #1\ekv@nil
104     }%
105   \long\def\ekv@strip@a##1\ekv@mark#1{\ekv@strip@b##1\ekv@mark}%
106   }
107 \ekv@strip{ }
108 \long\def\ekv@strip@b#1 \ekv@nil{\ekv@strip@c#1\ekv@nil}
109 \long\def\ekv@strip@c\ekv@mark#1\ekv@nil\ekv@mark#2\ekv@nil#3{#3{#1}}

```

(End definition for `\ekv@strip` and others.)

`\ekv@exparg` To reduce some code doublets while gaining some speed (and also as convenience for other packages in the family), it is often useful to expand the first token in a definition once. Let's define a wrapper for this.

`\ekv@exparg@` Also, to end a `\romannumeral` expansion, we want to use `\z@`, which is contained in both plain `TEX` and `LATEX`, but we use a private name for it to make it easier to spot and hence easier to manage.

```

110 \let\ekv@zero\z@
111 \long\def\ekv@exparg#1#2{\expandafter\ekv@exparg@\expandafter{#2}{#1}}
112 \long\def\ekv@exparg@#1#2{#2{#1}}%
113 \long\def\ekv@expargtwice#1#2{\expandafter\ekv@expargtwice@\expandafter{#2}{#1}}
114 \def\ekv@expargtwice@{\expandafter\ekv@exparg@\expandafter}

```

(End definition for `\ekv@exparg` and others.)

`\ekvcsvloop` An `\ekvcsvloop` will just loop over a `csv` list in a simple manner. First we split at active commas (gives better performance this way), next we have to check whether we're at the end of the list (checking for `\ekv@stop`). If not we go on splitting at commas of category other.

```

115 \begingroup
116 \def\ekvcsvloop#1{%
117 \endgroup
118 \long\def\ekvcsvloop##1##2%
119   {\ekv@csv@loop@active{##1}\ekv@mark##2#1\ekv@stop#1}

```

This does the same as `\ekv@csv@loop` but for active commas.

```

120 \long\def\ekv@csv@loop@active##1##2#1%
121   {%
122     \ekv@gobble@from@mark@to@stop##2\ekv@csv@loop@active@end\ekv@stop
123     \ekv@csv@loop{##1}##2,\ekv@stop,%
124   }%
125 \long\def\ekv@csv@loop@active@end
126   \ekv@stop
127   \ekv@csv@loop##1\ekv@mark\ekv@stop,\ekv@stop,%
128   {}%
129 }

```

Do the definitions with the weird catcode.

```

130 \catcode'\,=13
131 \ekvcsvloop,

```

(End definition for `\ekvcsvloop`, `\ekv@csv@loop@active`, and `\ekv@csv@loop@active@end`. These functions are documented on page 8.)

`\ekv@csv@loop`
`\ekv@csv@loop@do`
`\ekv@csv@loop@end`

We use temporary macros and an `\expandafter` chain to preexpand `\ekv@strip` here. After splitting at other commas we check again for end the end of the sublist, check for blank elements which should be ignored, and else strip spaces and execute the user code (protecting it from further expanding with `\unexpanded`).

```

132 \def\ekv@csv@loop#1%
133   {%
134     \long\def\ekv@csv@loop##1##2,%
135     {%
136       \ekv@gobble@from@mark@to@stop##2\ekv@csv@loop@end\ekv@stop
137       \ekv@ifblank@##2\ekv@nil\ekv@ifempty@B\ekv@csv@loop@blank
138       \ekv@ifempty@A\ekv@ifempty@B
139       #1\ekv@csv@loop@do{##1}%
140     }%
141   }
142 \expandafter\ekv@csv@loop\expandafter{\ekv@strip{#2}}
143 \long\def\ekv@csv@loop@do#1#2{\unexpanded{#2{#1}}\ekv@csv@loop{#2}\ekv@mark}
144 \def\ekv@csv@loop@end#1%
145   {%
146     \long\def\ekv@csv@loop@end
147     \ekv@stop
148     \ekv@ifblank@\ekv@mark\ekv@stop\ekv@nil\ekv@ifempty@B\ekv@csv@loop@blank
149     \ekv@ifempty@A\ekv@ifempty@B
150     #1\ekv@csv@loop@do##1%
151     {\ekv@csv@loop@active{##1}\ekv@mark}%
152   }
153 \expandafter\ekv@csv@loop@end\expandafter{\ekv@strip{\ekv@mark\ekv@stop}}
154 \long\expandafter\def\expandafter\ekv@csv@loop@blank
155   \expandafter\ekv@ifempty@A\expandafter\ekv@ifempty@B
156   \ekv@strip{\ekv@mark#1}\ekv@csv@loop@do#2%
157   {\ekv@csv@loop{#2}\ekv@mark}

```

(End definition for `\ekv@csv@loop`, `\ekv@csv@loop@do`, and `\ekv@csv@loop@end`.)

`\ekv@name`
`\ekv@name@set`
`\ekv@name@key`

The keys will all follow the same naming scheme, so we define it here.

```

158 \def\ekv@name@set#1{ekv#1{}}

```

```

159 \def\ekv@name@key#1{#1}}
160 \edef\ekv@name
161   {%
162     \unexpanded\expandafter{\ekv@name@set{#1}}%
163     \unexpanded\expandafter{\ekv@name@key{\detokenize{#2}}}%
164   }
165 \ekv@exparg{\def\ekv@name#1#2}{\ekv@name}

```

(End definition for `\ekv@name`, `\ekv@name@set`, and `\ekv@name@key`. These functions are documented on page 10.)

`\ekv@undefined@set` We can misuse the macro name we use to expandably store the set-name in a single token – since this increases performance drastically, especially for long set-names – to throw a more meaningful error message in case a set isn’t defined. The name of `\ekv@undefined@set` is a little bit misleading, as it is called in either case inside of `\csname`, but the result will be a control sequence with meaning `\relax` if the set is undefined, hence will break the `\csname` building the key-macro which will throw the error message.

```

166 \def\ekv@undefined@set#1{! expkv Error: Set ‘#1’ undefined.}

```

(End definition for `\ekv@undefined@set`.)

`\ekv@checkvalid` We place some restrictions on the allowed names, though, namely sets and keys are not allowed to be empty – blanks are fine (meaning set- or key-names consisting of spaces). The `\def\ekv@tmp` gobbles any TeX prefixes which would otherwise throw errors. This will, however, break the package if an `\outer` has been gobbled this way. I consider that good, because keys shouldn’t be defined `\outer` anyways.

```

167 \edef\ekv@checkvalid
168   {%
169     \unexpanded\expandafter{\ekv@ifempty{#1}}%
170     \unexpanded
171     {%
172       \def\ekv@tmp{}%
173       \errmessage{expkv Error: empty set name not allowed}%
174     }%
175     {%
176       \unexpanded\expandafter{\ekv@ifempty{#2}}%
177       \unexpanded
178       {%
179         \def\ekv@tmp{}%
180         \errmessage{expkv Error: empty key name not allowed}%
181       }%
182       \@secondoftwo
183     }%
184   }%
185   \unexpanded{\@gobble}%
186 }
187 \ekv@exparg{\protected\def\ekv@checkvalid#1#2}{\ekv@checkvalid}%

```

(End definition for `\ekv@checkvalid`.)

`\ekvifdefined` And provide user-level macros to test whether a key is defined.
`\ekvifdefinedNoVal`

```

189 \ekv@expargtwice{\def\ekvifdefined#1#2}%
190   {\expandafter\ekv@ifdefined\expandafter{\ekv@name{#1}{#2}}%

```



```

191 \ekv@expargtwice{\def\ekvifdefinedNoVal#1#2}%
192   {\expandafter\ekv@ifdefined\expandafter{\ekv@name{#1}{#2}N}}

```

(End definition for \ekvifdefined and \ekvifdefinedNoVal. These functions are documented on page 6.)

```

\ekvdef Set up the key defining macros \ekvdef etc. We use temporary macros to set these up
\ekvdefNoVal with a few expansions already done.
\ekvlet
\ekvletNoVal
\ekvletkv
\ekvletkvNoVal
\ekvdefunknown
\ekvdefunknownNoVal
193 \def\ekvdef#1#2#3#4%
194   {%
195     \protected\long\def\ekvdef##1##2##3%
196       {#1{\expandafter\def\csname#2\endcsname###1{##3}#3}}%
197     \protected\long\def\ekvdefNoVal##1##2##3%
198       {#1{\expandafter\def\csname#2N\endcsname{##3}#3}}%
199     \protected\def\ekvlet##1##2##3%
200       {#1{\expandafter\let\csname#2\endcsname##3#3}}%
201     \protected\def\ekvletNoVal##1##2##3%
202       {#1{\expandafter\let\csname#2N\endcsname##3#3}}%
203     \ekv@expargtwice{\protected\long\def\ekvdefunknown##1##2}%
204     {%
205       \romannumeral
206       \expandafter\ekv@exparg@\expandafter
207         {%
208           \expandafter\expandafter\expandafter
209           \def\expandafter\csname\ekv@name{##1}{-}u\endcsname###1###2{##2}%
210           #3%
211         }%
212       {\ekv@zero\ekv@checkvalid{##1}.}%
213     }%
214     \ekv@expargtwice{\protected\long\def\ekvdefunknownNoVal##1##2}%
215     {%
216       \romannumeral
217       \expandafter\ekv@exparg@\expandafter
218         {%
219           \expandafter\expandafter\expandafter
220           \def\expandafter\csname\ekv@name{##1}{-}uN\endcsname###1{##2}%
221           #3%
222         }%
223       {\ekv@zero\ekv@checkvalid{##1}.}%
224     }%
225     \protected\def\ekvletkv##1##2##3##4%
226     {%
227       #1%
228       {%
229         \expandafter\let\csname#2\expandafter\endcsname
230         \csname#4\endcsname
231         #3%
232       }%
233     }%
234     \protected\def\ekvletkvNoVal##1##2##3##4%
235     {%
236       #1%
237       {%
238         \expandafter\let\csname#2N\expandafter\endcsname
239         \csname#4N\endcsname

```

```

240         #3%
241     }%
242 }%
243 }
244 \edef\ekvdefNoVal
245 {%
246     {\unexpanded\expandafter{\ekv@checkvalid{#1}{#2}}}%
247     {\unexpanded\expandafter{\ekv@name{#1}{#2}}}%
248     {%
249         \unexpanded{\expandafter\ekv@defsetmacro\csname}%
250         \unexpanded\expandafter{\ekv@undefined@set{#1}\endcsname{#1}}%
251     }%
252     {\unexpanded\expandafter{\ekv@name{#3}{#4}}}%
253 }
254 \expandafter\ekvdef\ekvdefNoVal

```

(End definition for `\ekvdef` and others. These functions are documented on page 2.)

`\ekvredirectunknown` The redirection macros prepare the unknown function by looping over the provided list of sets and leaving a `\ekv@redirect@kv` or `\ekv@redirect@k` for each set. Only the first of these internals will receive the `<key>` and `<value>` as arguments.

```

\ekvredirectunknownNoVal
\ekv@defredirectunknown
\ekv@redirectunknown@aux
\ekv@redirectunknownNoVal@aux
255 \protected\def\ekvredirectunknown
256 {%
257     \ekv@defredirectunknown
258     \ekv@redirect@kv
259     \ekv@err@redirect@kv@notfound
260     {\long\ekvdefunknown}%
261     \ekv@redirectunknown@aux
262 }
263 \protected\def\ekvredirectunknownNoVal
264 {%
265     \ekv@defredirectunknown
266     \ekv@redirect@k
267     \ekv@err@redirect@k@notfound
268     \ekvdefunknownNoVal
269     \ekv@redirectunknownNoVal@aux
270 }
271 \protected\def\ekv@defredirectunknown#1#2#3#4#5#6%
272 {%
273     \begingroup
274     \edef\ekv@tmp
275     {%
276         \ekvcsvloop#1{#6}%
277         \unexpanded{#2}%
278         {\ekvcsvloop{}{#5,#6}}%
279     }%
280     \ekv@expargtwice
281     {\endgroup#3{#5}}%
282     {\expandafter#4\ekv@tmp\ekv@stop}%
283 }
284 \def\ekv@redirectunknown@aux#1{#1}{#2}
285 \def\ekv@redirectunknownNoVal@aux#1{#1}

```

(End definition for `\ekvredirectunknown` and others. These functions are documented on page 4.)

`\ekv@redirect@k` The redirect code works by some simple loop over all the sets, which we already preprocessed in `\ekv@defredirectunknown`. For some optimisation we blow this up a bit code wise, essentially, all this does is `\ekvifdefined` or `\ekvifdefinedNoVal` in each set, if there is a match gobble the remainder of the specified sets and execute the key macro, else go on with the next set (to which the `<key>` and `<value>` are forwarded).

First we set up some code which is different depending on `\lastnamedcs` being available or not. All this is stored in a temporary macro to have pre-expanded `\ekv@name` constellations ready.

```

286 \def\ekv@redirect@k##1##2##3##4%
287   {%
288     \ekv@if@lastnamedcs
289     {%
290       \def\ekv@redirect@k##1##2##3%
291       {%
292         \ifcsname#1\endcsname\ekv@redirect@k@a\fi
293         ##3{##1}%
294       }%
295       \def\ekv@redirect@k@a\fi{\fi\expandafter\ekv@redirect@k@b\lastnamedcs}%
296       \long\def\ekv@redirect@kv##1##2##3##4%
297       {%
298         \ifcsname#2\endcsname\ekv@redirect@kv@a\fi@gobble{##1}%
299         ##4{##1}{##2}%
300       }
301       \def\ekv@redirect@kv@a\fi@gobble
302       {\fi\expandafter\ekv@redirect@kv@b\lastnamedcs}%
303     }
304   {%
305     \def\ekv@redirect@k##1##2##3%
306     {%
307       \ifcsname#1\endcsname\ekv@redirect@k@a\fi\ekv@redirect@k@a@
308       #1\endcsname
309       ##3{##1}%
310     }%
311     \def\ekv@redirect@k@a@#3\endcsname{%
312     \def\ekv@redirect@k@a\fi\ekv@redirect@k@a@
313     {\fi\expandafter\ekv@redirect@k@b\csname}%
314     \long\def\ekv@redirect@kv##1##2##3##4%
315     {%
316       \ifcsname#2\endcsname\ekv@redirect@kv@a\fi\ekv@redirect@kv@a@
317       #2\endcsname{##1}%
318       ##4{##1}{##2}%
319     }
320     \long\def\ekv@redirect@kv@a@#4\endcsname##3{%
321     \def\ekv@redirect@kv@a\fi\ekv@redirect@kv@a@
322     {\fi\expandafter\ekv@redirect@kv@b\csname}%
323   }
324 }
  
```

The key name given to this loop will already be `\detokenized` by `\ekvset`, so we can safely remove the `\detokenize` here for some performance gain.

```

325 \def\ekv@redirect@kv##1\detokenize#2##3\ekv@stop{\unexpanded{##1##2##3}}
326 \edef\ekv@redirect@kv
327   {%
  
```

```

328   {\expandafter\ekv@redirect@kv\ekv@name{#2}-{#1}N\ekv@stop}%
329   {\expandafter\ekv@redirect@kv\ekv@name{#3}-{#2}\ekv@stop}%
330   {\expandafter\ekv@redirect@kv\ekv@name{#1}-{#2}N\ekv@stop}%
331   {\expandafter\ekv@redirect@kv\ekv@name{#1}-{#2}\ekv@stop}%
332 }

```

Everything is ready to make the real definitions.

```

333 \expandafter\ekv@redirect@k\ekv@redirect@kv

```

The remaining macros here are independent on `\lastnamedcs`, starting from the `@b` we know that there is a hash table entry, and get the macro as a parameter. We still have to test whether the macro is `\relax`, depending on the result of that test we have to either remove the remainder of the current test, or the remainder of the set list and invoke the macro.

```

334 \def\ekv@redirect@k@b#1%
335   {\ifx\relax#1\ekv@redirect@k@c\fi\ekv@redirect@k@d#1}
336 \def\ekv@redirect@k@c\fi\ekv@redirect@k@d#1{\fi}
337 \def\ekv@redirect@k@d#1#2\ekv@stop{#1}
338 \def\ekv@redirect@kv@b#1%
339   {\ifx\relax#1\ekv@redirect@kv@c\fi\ekv@redirect@kv@d#1}
340 \long\def\ekv@redirect@kv@c\fi\ekv@redirect@kv@d#1#2{\fi}
341 \long\def\ekv@redirect@kv@d#1#2#3\ekv@stop{#1{#2}}

```

(End definition for \ekv@redirect@k and others.)

`\ekv@defsetmacro` In order to enhance the speed the set name given to `\ekvset` will be turned into a control sequence pretty early, so we have to define that control sequence.

```

342 \edef\ekv@defsetmacro
343   {%
344     \unexpanded{\ifx#1\relax\edef#1##1}%
345     {%
346       \unexpanded\expandafter{\ekv@name@set{#2}}%
347       \unexpanded\expandafter{\ekv@name@key{##1}}%
348     }%
349     \unexpanded{\fi}%
350   }
351 \ekv@exparg{\protected\def\ekv@defsetmacro#1#2}{\ekv@defsetmacro}

```

(End definition for \ekv@defsetmacro.)

`\ekvifdefinedset`

```

352 \ekv@expargtwice{\def\ekvifdefinedset#1}%
353   {\expandafter\ekv@ifdefined\expandafter{\ekv@undefined@set{#1}}}

```

(End definition for \ekvifdefinedset. This function is documented on page 7.)

`\ekvset` Set up `\ekvset`, which should not be affected by active commas and equal signs. The equal signs are a bit harder to cope with and we'll do that later, but the active commas can be handled by just doing two comma-splitting loops one at a time and one at others. That's why we define `\ekvset` here with a temporary meaning just to set up the things with two different category codes. #1 will be a `_13` and #2 will be a `=_13`.

```

354 \begingroup
355 \def\ekvset#1#2{%
356 \endgroup
357 \ekv@exparg{\long\def\ekvset##1##2}%

```

```

358   {%
359     \expandafter\expandafter\expandafter
360     \ekv@set\expandafter\csname\ekv@undefined@set{##1}\endcsname
361     \ekv@mark##2#1\ekv@stop#1{}}%
362   }

```

(End definition for `\ekvset`. This function is documented on page 4.)

`\ekv@set` `\ekv@set` will split the `\langle key \rangle = \langle value \rangle` list at active commas. Then it has to check whether there were unprotected other commas and resplit there.

```

363 \long\def\ekv@set##1##2#1%
364   {%
Test whether we're at the end, if so invoke \ekv@endset,
365     \ekv@gobble@from@mark@to@stop##2\ekv@endset\ekv@stop
else go on with other commas.
366     \ekv@set@other##1##2,\ekv@stop,%
367   }

```

(End definition for `\ekv@set`.)

`\ekv@endset` `\ekv@endset` is a hungry little macro. It will eat everything that remains of `\ekv@set` and unbrace the sneaked stuff.

```

368 \long\def\ekv@endset
369   \ekv@stop\ekv@set@other##1\ekv@mark\ekv@stop,\ekv@stop,##2%
370   {##2}

```

(End definition for `\ekv@endset`.)

`\ekv@eq@other` `\ekv@eq@active` Splitting at equal signs will be done in a way that checks whether there is an equal sign and splits at the same time. This gets quite messy and the code might look complicated, but this is pretty fast (faster than first checking for an equal sign and splitting if one is found). The splitting code will be adapted for `\ekvset` and `\ekvparse` to get the most speed, but some of these macros don't require such adaptations. `\ekv@eq@other` and `\ekv@eq@active` will split the argument at the first equal sign and insert the macro which comes after the first following `\ekv@mark`. This allows for fast branching based on T_EX's argument grabbing rules and we don't have to split after the branching if the equal sign was there.

```

371 \long\def\ekv@eq@other##1=##2\ekv@mark##3{##3##1\ekv@stop\ekv@mark##2}
372 \long\def\ekv@eq@active##1#2##2\ekv@mark##3{##3##1\ekv@stop\ekv@mark##2}

```

(End definition for `\ekv@eq@other` and `\ekv@eq@active`.)

`\ekv@set@other` The macro `\ekv@set@other` is guaranteed to get only single `\langle key \rangle = \langle value \rangle` pairs.

```

373 \long\def\ekv@set@other##1##2,%
374   {%
First we test whether we're done.
375     \ekv@gobble@from@mark@to@stop##2\ekv@endset@other\ekv@stop
If not we split at the equal sign of category other.
376     \ekv@eq@other##2\ekv@nil\ekv@mark\ekv@set@eq@other@a
377     =\ekv@mark\ekv@set@eq@active

```

And insert the set name for the next recursion step of `\ekv@set@other`.

```
378     ##1%
379     \ekv@mark
380 }
```

(End definition for \ekv@set@other.)

`\ekv@set@eq@other@a` The first of these two macros runs the split-test for equal signs of category active. It will only be inserted if the `<key>=<value>` pair contained at least one equal sign of category other and `##1` will contain everything up to that equal sign.

```
381 \long\def\ekv@set@eq@other@a##1\ekv@stop
382   {%
383     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@set@eq@other@active
384     #2\ekv@mark\ekv@set@eq@other@b
385   }
```

The second macro will have been called by `\ekv@eq@active` if no active equal sign was found. All it does is remove the excess tokens of that test and forward the `<key>=<value>` pair to `\ekv@set@pair`. Normally we would have to also gobble an additional `\ekv@mark` after `\ekv@stop`, but this mark is needed to delimit `\ekv@set@pair`'s argument anyway, so we just leave it there.

```
386 \ekv@exparg
387   {%
388     \long\def\ekv@set@eq@other@b
389       ##1\ekv@nil\ekv@mark\ekv@set@eq@other@active\ekv@stop\ekv@mark
390       ##2\ekv@nil=\ekv@mark\ekv@set@eq@active
391   }%
392   {\ekv@strip{##1}{\expandafter\ekv@set@pair\detokenize}\ekv@mark##2\ekv@nil}
```

(End definition for \ekv@set@eq@other@a and \ekv@set@eq@other@b.)

`\ekv@set@eq@other@active` `\ekv@set@eq@other@active` will be called if the `<key>=<value>` pair was wrongly split on an equal sign of category other but has an earlier equal sign of category active. `##1` will be the contents up to the active equal sign and `##2` everything that remains until the first found other equal sign. It has to reinsert the equal sign and forward things to `\ekv@set@pair`.

```
393 \ekv@exparg
394   {%
395     \long\def\ekv@set@eq@other@active
396       ##1\ekv@stop##2\ekv@nil#2\ekv@mark
397     \ekv@set@eq@other@b\ekv@mark##3=\ekv@mark\ekv@set@eq@active
398   }%
399   {\ekv@strip{##1}{\expandafter\ekv@set@pair\detokenize}\ekv@mark##2=##3}
```

(End definition for \ekv@set@eq@other@active.)

`\ekv@set@eq@active` `\ekv@set@eq@active` will be called when there was no equal sign of category other in the `<key>=<value>` pair. It removes the excess tokens of the prior test and split-checks for an active equal sign.

```
400 \long\def\ekv@set@eq@active
401   ##1\ekv@nil\ekv@mark\ekv@set@eq@other@a\ekv@stop\ekv@mark
402   {%
403     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@set@eq@active@
404     #2\ekv@mark\ekv@set@noeq
405   }
```

If an active equal sign was found in `\ekv@set@eq@active` we'll have to pass the now split `<key>=<value>` pair on to `\ekv@set@pair`.

```
406 \ekv@exparg
407 {\long\def\ekv@set@eq@active###1\ekv@stop##2\ekv@nil#2\ekv@mark\ekv@set@noeq}%
408 {\ekv@strip{##1}{\expandafter\ekv@set@pair\detokenize}\ekv@mark##2\ekv@nil}
```

(End definition for \ekv@set@eq@active and \ekv@set@eq@active@.)

`\ekv@set@noeq` If no active equal sign was found by `\ekv@set@eq@active` there is no equal sign contained in the parsed list entry. In that case we have to check whether the entry is blank in order to ignore it (in which case we'll have to gobble the set-name which was put after these tests by `\ekv@set@other`). Else this is a `NoVal` key and the entry is passed on to `\ekv@set@key`.

```
409 \edef\ekv@set@noeq
410 {%
411   \unexpanded
412   {%
413     \ekv@ifblank###1\ekv@nil\ekv@ifempty@B\ekv@set@was@blank
414     \ekv@ifempty@A\ekv@ifempty@B
415   }%
416   \unexpanded\expandafter
417   {\ekv@strip{##1}{\expandafter\ekv@set@key\detokenize}\ekv@mark}%
418 }
419 \ekv@exparg
420 {%
421   \long\def\ekv@set@noeq
422     ##1\ekv@nil\ekv@mark\ekv@set@eq@active@\ekv@stop\ekv@mark
423   }%
424   {\ekv@set@noeq}
425   \expandafter\def\expandafter\ekv@set@was@blank
426   \expandafter\ekv@ifempty@A\expandafter\ekv@ifempty@B
427   \ekv@strip{\ekv@mark##1}##2\ekv@mark
428   {\ekv@set@other}
```

(End definition for \ekv@set@noeq.)

`\ekv@endset@other` All that's left for `\ekv@set@other` is the macro which breaks the recursion loop at the end. This is done by gobbling all the remaining tokens.

```
429 \long\def\ekv@endset@other
430   \ekv@stop
431   \ekv@eq@other\ekv@mark\ekv@stop\ekv@nil\ekv@mark\ekv@set@eq@other@a
432   =\ekv@mark\ekv@set@eq@active
433   {\ekv@set}
```

(End definition for \ekv@endset@other.)

`\ekvbreak` Provide macros that can completely stop the parsing of `\ekvset`, who knows what it'll be useful for.

`\ekvbreakPreSneak`
`\ekvbreakPostSneak`

```
434 \long\def\ekvbreak##1##2\ekv@stop#1##3{##1}
435 \long\def\ekvbreakPreSneak ##1##2\ekv@stop#1##3{##1##3}
436 \long\def\ekvbreakPostSneak##1##2\ekv@stop#1##3{##3##1}
```

(End definition for \ekvbreak, \ekvbreakPreSneak, and \ekvbreakPostSneak. These functions are documented on page 7.)

`\ekvsneak` One last thing we want to do for `\ekvset` is to provide macros that just smuggle stuff after `\ekvset`'s effects.

```
437 \long\def\ekvsneak##1##2\ekv@stop#1##3{##2\ekv@stop#1{##3##1}}
438 \long\def\ekvsneakPre##1##2\ekv@stop#1##3{##2\ekv@stop#1{##1##3}}
```

(End definition for `\ekvsneak` and `\ekvsneakPre`. These functions are documented on page 7.)

`\ekvparse` Additionally to the `\ekvset` macro we also want to provide an `\ekvparse` macro, that has the same scope as `\keyval_parse:NNn` from `expl3`. This is pretty analogue to the `\ekvset` implementation, we just put an `\unexpanded` here and there instead of other macros to stop the `\expanded` on our output. The `\unexpanded\expanded{. . .}` ensures that the material is in an alignment safe group at all time, and that it doesn't expand any further in an `\edef` or `\expanded` context.

```
439 \long\def\ekvparse##1##2##3%
440   {\unexpanded\expanded{\ekv@parse{##1}{##2}\ekv@mark##3#1\ekv@stop#1}}}
```

(End definition for `\ekvparse`. This function is documented on page 6.)

`\ekv@parse`

```
441 \long\def\ekv@parse##1##2##3#1%
442   {%
443     \ekv@gobble@from@mark@to@stop##3\ekv@endparse\ekv@stop
444     \ekv@parse@other{##1}{##2}##3,\ekv@stop,%
445   }
```

(End definition for `\ekv@parse`.)

`\ekv@endparse`

```
446 \long\def\ekv@endparse
447   \ekv@stop\ekv@parse@other##1\ekv@mark\ekv@stop,\ekv@stop,%
448   {}
```

(End definition for `\ekv@endparse`.)

`\ekv@parse@other`

```
449 \long\def\ekv@parse@other##1##2##3,%
450   {%
451     \ekv@gobble@from@mark@to@stop##3\ekv@endparse@other\ekv@stop
452     \ekv@eq@other##3\ekv@nil\ekv@mark\ekv@parse@eq@other@a
453     =\ekv@mark\ekv@parse@eq@active
454     {##1}{##2}%
455     \ekv@mark
456   }
```

(End definition for `\ekv@parse@other`.)

`\ekv@parse@eq@other@a`

`\ekv@parse@eq@other@b`

```
457 \long\def\ekv@parse@eq@other@a##1\ekv@stop
458   {%
459     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@other@active
460     #2\ekv@mark\ekv@parse@eq@other@b
461   }
462 \ekv@exparg
463   {%
```



```

464 \long\def\ekv@parse@eq@other@b
465   ##1\ekv@nil\ekv@mark\ekv@parse@eq@other@active\ekv@stop\ekv@mark
466   ##2\ekv@nil=\ekv@mark\ekv@parse@eq@active
467 }%
468 {\ekv@strip{##1}\ekv@parse@pair##2\ekv@nil}

```

(End definition for \ekv@parse@eq@other@a and \ekv@parse@eq@other@b.)

\ekv@parse@eq@other@active

```

469 \ekv@exparg
470 {%
471   \long\def\ekv@parse@eq@other@active
472     ##1\ekv@stop##2\ekv@nil#2\ekv@mark
473     \ekv@parse@eq@other@b\ekv@mark##3=\ekv@mark\ekv@parse@eq@active
474   }%
475   {\ekv@strip{##1}\ekv@parse@pair##2=##3}

```

(End definition for \ekv@parse@eq@other@active.)

\ekv@parse@eq@active

\ekv@parse@eq@active@

```

476 \long\def\ekv@parse@eq@active
477   ##1\ekv@nil\ekv@mark\ekv@parse@eq@other@a\ekv@stop\ekv@mark
478   {%
479     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@active@
480     #2\ekv@mark\ekv@parse@noeq
481   }
482 \ekv@exparg
483   {\long\def\ekv@parse@eq@active@##1\ekv@stop##2#2\ekv@mark\ekv@parse@noeq}%
484   {\ekv@strip{##1}\ekv@parse@pair##2}

```

(End definition for \ekv@parse@eq@active and \ekv@parse@eq@active@.)

\ekv@parse@noeq

```

485 \edef\ekv@parse@noeq
486   {%
487     \unexpanded
488     {%
489       \ekv@ifblank@##1\ekv@nil\ekv@ifempty@B\ekv@parse@was@blank
490       \ekv@ifempty@A\ekv@ifempty@B
491     }%
492     \unexpanded\expandafter{\ekv@strip{##1}\ekv@parse@key}%
493   }
494 \ekv@exparg
495   {%
496     \long\def\ekv@parse@noeq
497       ##1\ekv@nil\ekv@mark\ekv@parse@eq@active@\ekv@stop\ekv@mark
498     }%
499     {\ekv@parse@noeq}
500 \expandafter\def\expandafter\ekv@parse@was@blank
501   \expandafter\ekv@ifempty@A\expandafter\ekv@ifempty@B
502   \ekv@strip{\ekv@mark##1}\ekv@parse@key
503   {\ekv@parse@other}

```

(End definition for \ekv@parse@noeq.)

`\ekv@endparse@other`

```
504 \long\def\ekv@endparse@other
505   \ekv@stop
506   \ekv@eq@other\ekv@mark\ekv@stop\ekv@nil\ekv@mark\ekv@parse@eq@other@a
507   =\ekv@mark\ekv@parse@eq@active
508   {\ekv@parse}
(End definition for \ekv@endparse@other.)
```

`\ekv@parse@pair`
`\ekv@parse@pair@`

```
509 \ekv@exparg{\long\def\ekv@parse@pair##1##2\ekv@nil}%
510   {\ekv@strip{##2}\ekv@parse@pair@{##1}}
511 \long\def\ekv@parse@pair@##1##2##3##4%
512   {%
513     \unexpanded{##4{##2}{##1}}%
514     \ekv@parse@other{##3}{##4}%
515   }
(End definition for \ekv@parse@pair and \ekv@parse@pair@.)
```

`\ekv@parse@key`

```
516 \long\def\ekv@parse@key##1##2%
517   {%
518     \unexpanded{##2{##1}}%
519     \ekv@parse@other{##2}%
520   }
(End definition for \ekv@parse@key.)
```

Finally really setting things up with `\ekvset`'s temporary meaning:

```
521 }
522 \catcode'\,=13
523 \catcode'\==13
524 \ekvset,=
```

`\ekvsetSneaked` This macro can be defined just by expanding `\ekvsneak` once after expanding `\ekvset`. To expand everything as much as possible early on we use a temporary definition.

```
525 \edef\ekvsetSneaked
526   {%
527     \unexpanded{\ekvsneak{#2}}%
528     \unexpanded\expandafter{\ekvset{#1}{#3}}%
529   }
530 \ekv@expargtwice{\long\def\ekvsetSneaked#1#2#3{\ekvsetSneaked}
(End definition for \ekvsetSneaked. This function is documented on page 5.)
```

`\ekvchangeset` Provide a macro that is able to switch out the current `<set>` in `\ekvset`. This operation allows something similar to `pgfkeys's <key>/ .cd` mechanism. However this operation can be more expensive than `/ .cd` as we can't just redefine some token to reflect this, but have to switch out the set expandably, so this works similar to the `\ekvsneak` macros reading and reinserting things, but it only has to read and reinsert the remainder of the current key's replacement code.

```
531 \ekv@exparg{\def\ekvchangeset#1}%
532   {%
533     \expandafter\expandafter\expandafter
534     \ekv@changeset\expandafter\csname\ekv@undefined@set{#1}\endcsname\ekv@empty
535   }
```

(End definition for `\ekvchangeset`. This function is documented on page 7.)

`\ekv@changeset` This macro does the real change-out of `\ekvchangeset`. #2 will have a leading `\ekv@empty` so that braces aren't stripped accidentally, but that will not hurt and just expand to nothing in one step.

```
536 \long\def\ekv@changeset#1#2\ekv@set@other#3{#2\ekv@set@other#1}
```

(End definition for `\ekv@changeset`.)

`\ekv@set@pair` `\ekv@set@pair` gets invoked with the space and brace stripped and `\detokenized` key-name as its first, the value as the second, and the set name as the third argument. It provides tests for the key-macros and everything to be able to throw meaningful error messages if it isn't defined. We have two routes here, one if `\lastnamedcs` is defined and one if it isn't. The big difference is that if it is we can omit a `\csname` and instead just expand `\lastnamedcs` once to get the control sequence. If the macro is defined the value will be space and brace stripped and the key-macro called. Else branch into the error handling provided by `\ekv@set@pair`.

```
537 \ekv@if@lastnamedcs
538   {%
539     \long\def\ekv@set@pair#1\ekv@mark#2\ekv@nil#3%
540       {%
541         \ifcsname #3{#1}\endcsname\ekv@set@pair@a\fi\@secondoftwo
542         {#2}%
543         {%
544           \ifcsname #3{u}\endcsname\ekv@set@pair@a\fi\@secondoftwo
545           {#2}%
546           {%
547             \ekv@ifdefined{#3{#1}N}%
548             \ekv@err@noarg
549             \ekv@err@unknown
550             #3%
551           }%
552         {#1}%
553       }%
554     \ekv@set@other#3%
555   }
556 \def\ekv@set@pair@a\fi\@secondoftwo
557   {\fi\expandafter\ekv@set@pair@b\lastnamedcs}
558 }
559 {%
560 \long\def\ekv@set@pair#1\ekv@mark#2\ekv@nil#3%
561   {%
562     \ifcsname #3{#1}\endcsname
563     \ekv@set@pair@a\fi\ekv@set@pair@c#3{#1}\endcsname
564     {#2}%
565     {%
566       \ifcsname #3{u}\endcsname
567       \ekv@set@pair@a\fi\ekv@set@pair@c#3{u}\endcsname
568       {#2}%
569       {%
570         \ekv@ifdefined{#3{#1}N}%
571         \ekv@err@noarg
572         \ekv@err@unknown
```

```

573             #3%
574             }%
575             {#1}%
576         }%
577     \ekv@set@other#3%
578 }
579 \def\ekv@set@pair@a\fi\ekv@set@pair@c{\fi\expandafter\ekv@set@pair@b\csname}
580 \long\def\ekv@set@pair@c#1\endcsname#2#3{#3}
581 }
582 \long\def\ekv@set@pair@b#1%
583 {%
584     \ifx#1\relax
585         \ekv@set@pair@e
586     \fi
587     \ekv@set@pair@d#1%
588 }
589 \ekv@exparg{\long\def\ekv@set@pair@d#1#2#3{\ekv@strip{#2}#1}
590 \long\def\ekv@set@pair@e\fi\ekv@set@pair@d#1#2#3{\fi#3}

```

(End definition for \ekv@set@pair and others.)

\ekv@set@key Analogous to \ekv@set@pair, \ekv@set@key builds the NoVal key-macro and provides an error-branch. \ekv@set@key@a will test whether the key-macro is defined and if so call it, else the errors are thrown.

```

\ekv@set@key@a
\ekv@set@key@b
\ekv@set@key@c
591 \ekv@if@lastnamedcs
592 {%
593     \long\def\ekv@set@key#1\ekv@mark#2%
594     {%
595         \ifcsname #2{#1}N\endcsname\ekv@set@key@a\fi\@firstofone
596         {%
597             \ifcsname #2{uN}\endcsname\ekv@set@key@a\fi\@firstofone
598             {%
599                 \ekv@ifdefined{#2{#1}}%
600                 \ekv@err@reqval
601                 \ekv@err@unknown
602                 #2%
603             }%
604         }%
605     }%
606     \ekv@set@other#2%
607 }
608 \def\ekv@set@key@a\fi\@firstofone{\fi\expandafter\ekv@set@key@b\lastnamedcs}
609 }
610 {%
611     \long\def\ekv@set@key#1\ekv@mark#2%
612     {%
613         \ifcsname #2{#1}N\endcsname
614         \ekv@set@key@a\fi\ekv@set@key@c#2{#1}N\endcsname
615         {%
616             \ifcsname #2{uN}\endcsname
617             \ekv@set@key@a\fi\ekv@set@key@c#2{uN}\endcsname
618             {%
619                 \ekv@ifdefined{#2{#1}}%
620                 \ekv@err@reqval

```

```

621         \ekv@err@unknown
622         #2%
623     }%
624     {#1}%
625 }%
626 \ekv@set@other#2%
627 }
628 \def\ekv@set@key@a\fi\ekv@set@key@c{\fi\expandafter\ekv@set@key@b\csname}
629 \long\def\ekv@set@key@c#1N\endcsname#2{#2}
630 }
631 \long\def\ekv@set@key@b#1%
632 {%
633     \ifx#1\relax
634         \ekv@fi@secondoftwo
635     \fi
636     \@firstoftwo#1%
637 }

```

(End definition for \ekv@set@key and others.)

\ekvsetdef Provide a macro to define a shorthand to use \ekvset on a specified `<set>`. To gain the maximum speed \ekvset is expanded twice by \ekv@exparg so that during runtime the macro storing the set name is already built and one \expandafter doesn't have to be used.

```

638 \ekv@expargtwice{\protected\def\ekvsetdef#1#2}%
639 {%
640     \romannumeral
641     \ekv@exparg{\ekv@zero\ekv@exparg{\long\def#1##1}}%
642     {\ekvset{#2}{##1}}%
643 }

```

(End definition for \ekvsetdef. This function is documented on page 5.)

\ekvsetSneakeddef And do the same for \ekvsetSneaked in the two possible ways, with a fixed sneaked argument and with a flexible one.

\ekvsetdefSneaked

```

644 \ekv@expargtwice{\protected\def\ekvsetSneakeddef#1#2}%
645 {%
646     \romannumeral
647     \ekv@exparg{\ekv@zero\ekv@exparg{\long\def#1##1##2}}%
648     {\ekvsetSneaked{#2}{##1}{##2}}%
649 }
650 \ekv@expargtwice{\protected\def\ekvsetdefSneaked#1#2#3}%
651 {%
652     \romannumeral
653     \ekv@exparg{\ekv@zero\ekv@exparg{\long\def#1##1}}%
654     {\ekvsetSneaked{#2}{#3}{##1}}%
655 }

```

(End definition for \ekvsetSneakeddef and \ekvsetdefSneaked. These functions are documented on page 5.)

\ekv@alignsafe These macros protect the usage of ampersands inside of alignment contexts.

\ekv@endalignsafe

```

656 \begingroup
657 \catcode'\^^@=2
658 \@firstofone{\endgroup}

```

```

659 \def\ekv@alignsafe{\romannumeral\iffalse{\fi'^^@ }
660 }
661 \def\ekv@endalignsafe{\ifnum'={\ekv@zero}\fi}

```

(End definition for `\ekv@alignsafe` and `\ekv@endalignsafe`.)

`\ekvoptarg`
`\ekvoptargTF`

Provide macros to expandably collect an optional argument in brackets. The macros here are pretty simple in nature compared to `xparse`'s possibilities (they don't care for nested bracket levels).

We start with a temporary definition to pre-expand `\ekv@alignsafe` (will be #1) and `\ekv@endalignsafe` (will be #2). As `\ekv@alignsafe` starts with a `\romannumeral` we use that to also control the number of steps needed instead of adding another `\romannumeral`. For this we have to remove the space token from the end of `\ekv@alignsafe`'s definition.

```

662 \begingroup
663 \def\ekvoptarg#1#2{%
664 \endgroup

```

The real definition starts an expansion context and afterwards grabs the arguments. #1 will be the next step, #2 the default value, and #3 might be an opening bracket, or the mandatory argument. We check for the opening bracket, if it is found grab the optional argument, else leave #1{#2} in the input stream after ending the expansion context.

```

665 \def\ekvoptarg{#1\ekv@optarg@a}
666 \long\def\ekv@optarg@a##1##2##3%
667   {%
668     \ekv@optarg@if\ekv@mark##3\ekv@mark\ekv@optarg@b\ekv@mark[\ekv@mark
669     #2%
670     \@firstofone{ ##1}{##2}{##3}%
671   }%

```

The other variant of this will do roughly the same. Here, #1 will be the next step if an optional argument is found, #2 the next step else, and #3 might be the opening bracket or mandatory argument.

```

672 \def\ekvoptargTF{#1\ekv@optargTF@a}
673 \long\def\ekv@optargTF@a##1##2##3%
674   {%
675     \ekv@optarg@if\ekv@mark##3\ekv@mark\ekv@optargTF@b{##1}\ekv@mark[\ekv@mark
676     #2%
677     \@firstofone{ ##2}{##3}%
678   }

```

The two macros to grab the optional argument have to remove the remainder of the test and the wrong next step as well as grabbing the argument.

```

679 \long\def\ekv@optarg@b\ekv@mark[\ekv@mark\ifnum'##1\fi\@firstofone##2##3##4##5)%
680   {##2{##5}}
681 \long\def\ekv@optargTF@b
682   ##1\ekv@mark[\ekv@mark\ifnum'##2\fi\@firstofone##3##4##5)%
683   {##2 ##1{##5}}
684 }

```

Do the definitions and add the test macro. We use `\ekv@strip` to remove the trailing space from the definition of `\ekv@alignsafe`.

```

685 \ekv@exparg
686   {%
687     \expandafter\ekv@strip\expandafter

```

```

688     {\expandafter\ekv@mark\ekv@alignsafe}%
689     \ekvoptarg
690   }%
691   \ekv@endalignsafe
692 \long\def\ekv@optarg@if#1\ekv@mark[\ekv@mark{}

```

(End definition for `\ekvoptarg` and `\ekvoptargTF`. These functions are documented on page 8.)

`\ekv@err@collect` `\ekv@err@cleanup` `\ekv@err` Since `\ekvset` is fully expandable as long as the code of the keys is (which is unlikely) we want to somehow throw expandable errors, in our case via a runaway argument (to my knowledge the first version of this method was implemented by Jean-François Burnol, many thanks to him). The first step is to ensure that the second argument (which might contain user input) doesn't contain tokens we use as delimiters (in this case `\par`), this will be done by the front facing macro `\ekv@err`. But first we set some other things up.

We use a temporary definition for `\ekv@err` to get multiple consecutive spaces. Then we set up the macro that will collect the error and the macro that will throw the error. The latter will have an unreasonable long name. This way we can convey more information. Though the information in the macro name is static and has to be somewhat general to fit every occurrence. The important bit is that the long named macro has a delimited argument and is short which will throw the error at the `\par` at the end of `\ekv@err@collect`. This macro has the drawback that it will only print nicely if the `\newlinechar` is `^^J`.

```

693 \def\ekv@err@cleanup\par{}
694 \def\ekv@err@collect#1%
695   {%
696     \def\ekv@err@collect##1\par##2%
697     {%
698       \expandafter
699       \ekv@err@cleanup
700       #1! ##2 Error: ##1\par
701     }%
702     \def#1##1\thanks@jfbu{}%
703   }
704 \def\ekv@err{ }
705 \expandafter\ekv@err@collect\csname <an-expandable-macro>^^J%
706   completed due to above exception. \ekv@err If the error^^J%
707   summary is \ekv@err not comprehensible \ekv@err see the package^^J%
708   documentation.^^J%
709   I will try to recover now. \ekv@err If you're in inter-^^J%
710   active mode hit <return> \ekv@err at the ? prompt and I^^J%
711   continue hoping recovery\endcsname
712 \long\def\ekv@err#1#2{\expandafter\ekv@err@collect\detokenize{#2}\par{#1}}

```

(End definition for `\ekv@err`, `\ekv@err@collect`, and `\ekv@err@cleanup`. These functions are documented on page 9.)

`\ekv@err` We define a shorthand to throw errors in `expkV`.

```

713 \ekv@exparg{\long\def\ekv@err#1}{\ekv@err{expkv}{#1}}

```

(End definition for `\ekv@err`.)

```

\ekv@err@common Now we can use \ekv@err to set up some error messages so that we can later use those
\ekv@err@common@ instead of the full strings.
\ekv@err@unknown 714 \long\def\ekv@err@common #1#2{\expandafter\ekv@err@common@\string#2{#1}}
\ekv@err@noarg 715 \ekv@exparg{\long\def\ekv@err@common@#1'#2' #3.#4#5}%
\ekv@err@reqval 716 {\ekv@err{#4 '#5' in set '#2'}}
717 \ekv@exparg{\long\def\ekv@err@unknown#1}{\ekv@err@common{unknown key}{#1}}
718 \ekv@exparg{\long\def\ekv@err@noarg #1}{\ekv@err@common{unwanted value for}{#1}}
719 \ekv@exparg{\long\def\ekv@err@reqval #1}{\ekv@err@common{missing value for}{#1}}
720 \ekv@exparg{\long\def\ekv@err@redirect@k@notfound#1#2#3\ekv@stop}%
721 {\ekv@err{no key '#2' in sets #3}}
722 \ekv@exparg{\def\ekv@err@redirect@k@notfound#1#2\ekv@stop}%
723 {\ekv@err{no NoVal key '#1' in sets #2}}

(End definition for \ekv@err@common and others.)
Now everything that's left is to reset the category code of @.
724 \catcode'\@=\ekv@tmp

```


Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	N
\wedge	\backslash noexpand 20
	P
	\backslash par 693, 696, 700, 712
E	T
\backslash ekvbreak 7, <u>434</u>	TeX and L ^A T _E X 2 _ε commands:
\backslash ekvbreakPostSneak 7, <u>434</u>	\backslash @firstofone 40, 59, 60, 63,
\backslash ekvbreakPreSneak 7, <u>434</u>	595, 597, 608, 658, 670, 677, 679, 682
\backslash ekvchangeset 7, <u>531</u>	\backslash @firstoftwo 40, 58, 80, 89, 93, 636
\backslash ekvcsvloop 8, <u>115</u> , 276, 278	\backslash @gobble 40, 186, 298, 301
\backslash ekvDate 6, 4, 8, <u>23</u>	\backslash @secondoftwo 40,
\backslash ekvdef 2, <u>193</u>	54, 57, 68, 73, 74, 183, 541, 544, 556
\backslash ekvdefNoVal 3, <u>193</u>	\backslash ekv@alignsafe <u>656</u> , 688
\backslash ekvdefunknown 3, <u>193</u> , 260	\backslash ekv@changeset 534, <u>536</u>
\backslash ekvdefunknownNoVal 3, <u>193</u> , 268	\backslash ekv@checkvalid <u>167</u> , 212, 223, 246
\backslash ekverr 9, <u>693</u> , 713	\backslash ekv@csv@loop 123, 127, <u>132</u>
\backslash ekvifdefined 6, <u>189</u>	\backslash ekv@csv@loop@active <u>115</u> , 151
\backslash ekvifdefinedNoVal 6, <u>189</u>	\backslash ekv@csv@loop@active@end <u>115</u>
\backslash ekvifdefinedset 7, <u>352</u>	\backslash ekv@csv@loop@blank 137, 148, 154
\backslash ekvlet 3, <u>193</u>	\backslash ekv@csv@loop@do <u>132</u>
\backslash ekvletkv 3, <u>193</u>	\backslash ekv@csv@loop@end <u>132</u>
\backslash ekvletkvNoVal 3, <u>193</u>	\backslash ekv@defredirectunknown <u>255</u>
\backslash ekvletNoVal 3, <u>193</u>	\backslash ekv@defsetmacro 249, <u>342</u>
\backslash ekvoptarg 8, <u>662</u>	\backslash ekv@empty 39, 534
\backslash ekvoptargTF 8, <u>662</u>	\backslash ekv@endalignsafe <u>656</u> , 691
\backslash ekvparse 6, <u>439</u>	\backslash ekv@endparse 443, <u>446</u>
\backslash ekvredirectunknown 4, <u>255</u>	\backslash ekv@endparse@other 451, <u>504</u>
\backslash ekvredirectunknownNoVal 4, <u>255</u>	\backslash ekv@endset 365, <u>368</u>
\backslash ekvset 4, <u>354</u> , 524, 528, 642	\backslash ekv@endset@other 375, <u>429</u>
\backslash ekvsetdef 5, <u>638</u>	\backslash ekv@eq@active <u>371</u> , 383, 403, 459, 479
\backslash ekvsetdefSneaked 5, <u>644</u>	\backslash ekv@eq@other <u>371</u> , 376, 431, 452, 506
\backslash ekvsetSneaked 5, <u>525</u> , 648, 654	\backslash ekv@err <u>713</u> , 716, 721, 723
\backslash ekvsetSneakeddef 5, <u>644</u>	\backslash ekv@err@cleanup <u>693</u>
\backslash ekvsneak 7, <u>437</u> , 527	\backslash ekv@err@collect <u>693</u>
\backslash ekvsneakPre 7, <u>437</u>	\backslash ekv@err@common <u>714</u>
\backslash ekvtmpa 14, 18	\backslash ekv@err@common@ <u>714</u>
\backslash ekvtmpb 15, 18	\backslash ekv@err@noarg 548, 571, <u>714</u>
\backslash ekvVersion 6, 4, 8, <u>23</u>	\backslash ekv@err@redirect@k@notfound 267, <u>722</u>
	\backslash ekv@err@redirect@kv@notfound 259, <u>720</u>
I	\backslash ekv@err@reqval 600, 620, <u>714</u>
\backslash iffalse 659	
\backslash ifnum 661, 679, 682	

<code>\ekv@err@unknown</code>	549, 572, 601, 621, <u>714</u>	<code>\ekv@name@set</code>	10, <u>158</u> , 346
<code>\ekv@exparg</code>	. <u>110</u> , 165, 188, 351, 357, 386, 393, 406, 419, 462, 469, 482, 494, 509, 531, 589, 641, 647, 653, 685, 713, 715, 717, 718, 719, 720, 722	<code>\ekv@nil</code> 67, 101, 103, 108, 109, 137, 148, 376, 383, 389, 390, 392, 396, 401, 403, 407, 408, 413, 422, 431, 452, 459, 465, 466, 468, 472, 477, 479, 489, 497, 506, 509, 539, 560
<code>\ekv@exparg@</code> <u>110</u> , 206, 217	<code>\ekv@optarg@a</code> 665, 666
<code>\ekv@expargtwice</code>	... <u>110</u> , 189, 191, 203, 214, 280, 352, 530, 638, 644, 650	<code>\ekv@optarg@b</code> 668, 679
<code>\ekv@expargtwice@</code> <u>110</u>	<code>\ekv@optarg@if</code> 668, 675, 692
<code>\ekv@fi@firstofone</code> <u>40</u>	<code>\ekv@optargTF@a</code> 672, 673
<code>\ekv@fi@firstoftwo</code> <u>40</u>	<code>\ekv@optargTF@b</code> 675, 681
<code>\ekv@fi@gobble</code> <u>40</u>	<code>\ekv@parse</code> 440, <u>441</u> , 508
<code>\ekv@fi@secondoftwo</code>	<u>40</u> , 78, 87, 93, 634	<code>\ekv@parse@eq@active</code> 453, 466, 473, <u>476</u> , 507
<code>\ekv@gobble@from@mark@to@stop</code> <u>40</u> , 122, 136, 365, 375, 443, 451	<code>\ekv@parse@eq@active@</code> <u>476</u> , 497
<code>\ekv@gobble@mark</code> <u>40</u>	<code>\ekv@parse@eq@other@a</code> 452, <u>457</u> , 477, 506
<code>\ekv@gobbleto@stop</code> <u>40</u>	<code>\ekv@parse@eq@other@active</code> 459, 465, <u>469</u>
<code>\ekv@if@lastnamedcs</code> 28, 71, 288, 537, 591	<code>\ekv@parse@eq@other@b</code> 457, 473
<code>\ekv@ifblank</code> <u>65</u>	<code>\ekv@parse@key</code> 492, 502, <u>516</u>
<code>\ekv@ifblank@</code>	... <u>65</u> , 137, 148, 413, 489	<code>\ekv@parse@noeq</code> 480, 483, <u>485</u>
<code>\ekv@ifdef@</code> 73, 74, 86, 91	<code>\ekv@parse@other</code> 444, 447, <u>449</u> , 503, 514, 519
<code>\ekv@ifdef@false</code> 86, 91, 92	<code>\ekv@parse@pair</code>	... 468, 475, 484, <u>509</u>
<code>\ekv@ifdefined</code> <u>71</u> , 190, 192, 353, 547, 570, 599, 619	<code>\ekv@parse@pair@</code> <u>509</u>
<code>\ekv@ifempty</code> <u>51</u> , 169, 176	<code>\ekv@parse@was@blank</code> 489, 500
<code>\ekv@ifempty@</code> <u>51</u> , 70	<code>\ekv@redirect@k</code> <u>266</u> , <u>286</u>
<code>\ekv@ifempty@A</code> 53, 54, 56, 57, 58, 59, 60, 63, 68, 70, 138, 149, 155, 414, 426, 490, 501	<code>\ekv@redirect@k@a</code> <u>286</u>
<code>\ekv@ifempty@B</code> 53, 54, 56, 57, 58, 59, 60, 63, 67, 68, 137, 138, 148, 149, 155, 413, 414, 426, 489, 490, 501	<code>\ekv@redirect@k@a@</code> <u>286</u>
<code>\ekv@ifempty@false</code> <u>51</u>	<code>\ekv@redirect@k@b</code> <u>286</u>
<code>\ekv@ifempty@true</code> <u>51</u> , 67	<code>\ekv@redirect@k@c</code> <u>286</u>
<code>\ekv@ifempty@true@F</code> <u>51</u>	<code>\ekv@redirect@k@d</code> <u>286</u>
<code>\ekv@ifempty@true@F@gobble</code> <u>51</u>	<code>\ekv@redirect@kv</code> 258, <u>286</u>
<code>\ekv@ifempty@true@F@gobbletwo</code>	... <u>51</u>	<code>\ekv@redirect@kv@a</code> <u>286</u>
<code>\ekv@mark</code> 48, 50, 70, 102, 105, 109, 119, 127, 143, 148, 151, 153, 156, 157, 361, 369, 371, 372, 376, 377, 379, 383, 384, 389, 390, 392, 396, 397, 399, 401, 403, 404, 407, 408, 417, 422, 427, 431, 432, 440, 447, 452, 453, 455, 459, 460, 465, 466, 472, 473, 477, 479, 480, 483, 497, 502, 506, 507, 539, 560, 593, 611, 668, 675, 679, 682, 688, 692	<code>\ekv@redirect@kv@a@</code> <u>286</u>
<code>\ekv@name</code> 10, <u>158</u> , 190, 192, 209, 220, 247, 252, 328, 329, 330, 331	<code>\ekv@redirect@kv@b</code> <u>286</u>
<code>\ekv@name@key</code> 10, <u>158</u> , 347	<code>\ekv@redirect@kv@c</code> <u>286</u>
		<code>\ekv@redirect@kv@d</code> <u>286</u>
		<code>\ekv@redirectunknown@aux</code> <u>255</u>
		<code>\ekv@redirectunknown@noval@aux</code>	... <u>255</u>
		<code>\ekv@set</code> 360, <u>363</u> , 433
		<code>\ekv@set@eq@active</code> 377, 390, 397, <u>400</u> , 432
		<code>\ekv@set@eq@active@</code> <u>400</u> , 422
		<code>\ekv@set@eq@other@a</code>	376, <u>381</u> , 401, 431
		<code>\ekv@set@eq@other@active</code>	383, 389, <u>393</u>
		<code>\ekv@set@eq@other@b</code> <u>381</u> , 397
		<code>\ekv@set@key</code> 417, <u>591</u>
		<code>\ekv@set@key@a</code> <u>591</u>
		<code>\ekv@set@key@b</code> <u>591</u>

<code>\ekv@set@key@c</code>	591	375 , 381 , 389 , 396 , 401 , 407 , 422 ,
<code>\ekv@set@noeq</code>	404 , 407 , 409	430 , 431 , 434 , 435 , 436 , 437 , 438 ,
<code>\ekv@set@other</code>	366 ,	440 , 443 , 444 , 447 , 451 , 457 , 465 ,
	369 , 373 , 428 , 536 , 554 , 577 , 606 , 626	472 , 477 , 483 , 497 , 505 , 506 , 720 , 722
<code>\ekv@set@pair</code>	392 , 399 , 408 , 537	<code>\ekv@strip</code>
<code>\ekv@set@pair@a</code>	537	96 , 142 ,
<code>\ekv@set@pair@b</code>	537	153 , 156 , 392 , 399 , 408 , 417 , 427 ,
<code>\ekv@set@pair@c</code>	537	468 , 475 , 484 , 492 , 502 , 510 , 589 , 687
<code>\ekv@set@pair@d</code>	537	<code>\ekv@strip@a</code>
<code>\ekv@set@pair@e</code>	537	96
<code>\ekv@set@was@blank</code>	413 , 425	<code>\ekv@strip@b</code>
<code>\ekv@stop</code>	49 , 50 , 119 , 122 ,	96
	123 , 126 , 127 , 136 , 147 , 148 , 153 ,	<code>\ekv@strip@c</code>
	282 , 325 , 328 , 329 , 330 , 331 , 337 ,	96
	341 , 361 , 365 , 366 , 369 , 371 , 372 ,	<code>\ekv@tmp</code>
		1 , 172 , 180 , 274 , 282 , 724
		<code>\ekv@tmpa</code>
		29 , 31
		<code>\ekv@tmpb</code>
		30 , 31
		<code>\ekv@undefined@set</code>
		166 , 250 , 353 , 360 , 534
		<code>\ekv@zero</code> 110 , 212 , 223 , 641 , 647 , 653 , 661
		<code>\thanks@jfbu</code>
		702