# expkv-cs

## define expandable ⟨*key*⟩=⟨*value*⟩ macros using expkv

Jonathan P. Spratte*

2020-08-08 v0.4

**Abstract**

expkv-cs provides two small interfaces to define expandable ⟨*key*⟩=⟨*value*⟩ macros using expkv. It therefore lowers the entrance boundary to expandable ⟨*key*⟩=⟨*value*⟩ macros. The stylised name is expkv-cs but the files use `expkv-cs`, this is due to CTAN-rules which don't allow | in package names since that is the pipe symbol in *nix shells.

## Contents



*jspratte@yahoo.de

# 1 Documentation

The expkv package enables the new possibility of creating ⟨*key*⟩=⟨*value*⟩ macros which are fully expandable. The creation of such macros is however cumbersome for the average user. expkv-cs tries to step in here. It provides interfaces to define ⟨*key*⟩=⟨*value*⟩ macros without worrying too much about the implementation. In case you're wondering now, the `cs` in expkv-cs stands for control sequence, because `def` was already taken by expkv-def and "control sequence" is the term D. E. Knuth used in his TEXbook for named commands hence macros (though he also used the term "macro"). So expkv-cs defines control sequences for and with expkv.

There are two different approaches supported by this package. The first is splitting the keys up into individual arguments, the second is providing all the keys as a single argument to the underlying macro and getting an individual ⟨*value*⟩ by using a hash. Well, actually there is no real hash, just some markers which are parsed, but this shouldn't be apparent to the user, the behaviour matches that of a hash-table.

In addition to these two methods of defining a macro with primary keys a way to define secondary keys, which can reference the primary ones, is provided. These secondary keys don't correspond to an argument or an entry in the hash table directly but might come in handy for the average use case. Each macro has its own set of primary and secondary keys.

A word of advice you should consider: If your macro doesn't have to be expandable (and often it doesn't) don't use expkv-cs. The interface has some overhead (though it still can be considered fast – check subsection 1.4) and the approach has its limits in versatility. If you don't need to be expandable, you should consider either defining your keys manually using expkv or using expkv-def for convenience. Or you resort to another ⟨*key*⟩=⟨*value*⟩ interface.

expkv-cs is usable as generic code and as a LATEX package. It'll automatically load expkv in the same mode as well. To use it, just use one of

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

## 1.1 Define Macros and Primary Keys

All macros defined with expkv-cs have to be previously undefined or have the `\meaning` of `\relax`. This is necessary as there is no way to undefine keys once they are set up (neither expkv nor expkv-cs keep track of defined keys) – so to make sure there are no conflicts only new definitions are allowed (that's not the case for individual keys, only for frontend macros).

### 1.1.1 Primary Keys

In the following descriptions there will be one argument named ⟨*primary keys*⟩. This argument should be a ⟨*key*⟩=⟨*value*⟩ list where each ⟨*key*⟩ will be one primary key and ⟨*value*⟩ the associated initial value. By default all keys are defined short, but you can define long keys by prefixing ⟨*key*⟩ with `long` (*e.g.*, `long name=Jonathan P. Spratte`). You only need `long` if the key should be able to take a `\par` token. Note however that `long` keys are a microscopic grain faster (due to some internals of expkv-cs). Only if at least one of the keys was `long` the ⟨*cs*⟩ in the following defining macros will be `\long`. For obvious reasons there is no possibility to define a macro or key as `\protected`.

At the moment expkvics doesn't require any internal keys, but I can't foresee whether this will be the case in the future as well, as it might turn out that some features I deem useful can't be implemented without such internal keys. Because of this, please don't use key names starting with `EKVC|` as that should be the private name space.

### 1.1.2 Split

The split variants will provide the key values as separate arguments. This limits the number of keys for which this is truly useful.

`\ekvcSplit` `\ekvcSplit⟨cs⟩{⟨primary keys⟩}{⟨definition⟩}`

This defines ⟨*cs*⟩ to be a macro taking one mandatory argument which should contain a ⟨*key*⟩=⟨*value*⟩ list. The ⟨*primary keys*⟩ will be defined for this macro (see subsubsection 1.1.1). The ⟨*definition*⟩ is the code that will be executed. You can access the ⟨*value*⟩ of a ⟨*key*⟩ by using a macro parameter from `#1` to `#9`. The order of the macro parameters will be the order provided in the ⟨*primary keys*⟩ list (so `#1` is the ⟨*value*⟩ of the key defined first). With `\ekvcSplit` you can define macros using at most nine primary keys.

`\ekvcSplitAndForward` `\ekvcSplitAndForward⟨cs⟩{⟨after⟩}{⟨primary keys⟩}`

This defines ⟨*cs*⟩ to be a macro taking one mandatory argument which should contain a ⟨*key*⟩=⟨*value*⟩ list. You can use as many primary keys as you want with this. The primary keys will be forwarded to ⟨*after*⟩ as braced arguments (as many as necessary for your primary keys). The order of the braced arguments will be the order of your primary key definitions. In ⟨*after*⟩ you can use just a single control sequence, or some arbitrary stuff which will be left in the input stream before your braced values (with one set of braces stripped from ⟨*after*⟩), so both of the following would be fine:

*\ekvcSplitAndForward\foo\foo@aux{keyA = A, keyB = B}*
*\ekvcSplitAndForward\foo{\foo@aux{more args}}{keyA = A, keyB = B}*

`\ekvcSplitAndUse` `\ekvcSplitAndUse⟨cs⟩{⟨primary keys⟩}`

This will roughly do the same as `\ekvcSplitAndForward`, but instead of specifying what will be used after splitting the keys, ⟨*cs*⟩ will use what follows the ⟨*key*⟩=⟨*value*⟩ list. So its syntax will be

⟨*cs*⟩{⟨*key*⟩=⟨*value*⟩, ...}{⟨*after*⟩}

### 1.1.3 Hash

The hash variants will provide the key values as a single argument in which you can access specific values using a special macro. The implementation might be more convenient and scale better, *but* it is much slower (for a primitive macro with a single key benchmarking was almost 1.7 times slower, the root of which being the key access with `\ekvcValue`, not the parsing, and for a key access using `\ekvcValueFast` it was still about 1.2 times slower). So if your macro uses less than ten primary keys, you should most likely use the split approach.

\ekvcHash `\ekvcHash⟨cs⟩{⟨primary keys⟩}{⟨definition⟩}`

This defines ⟨*cs*⟩ to be a macro taking one mandatory argument which should contain a ⟨*key*⟩=⟨*value*⟩ list. You can use as many primary keys as you want. The primary keys will be forwarded as a single argument containing every key to the underlying macro. The underlying macro is defined as ⟨*definition*⟩, in which you can access the ⟨*value*⟩ of a ⟨*key*⟩ by using `\ekvcValue{⟨key⟩}{#1}`.

\ekvcHashAndForward `\ekvcHashAndForward⟨cs⟩{⟨after⟩}{⟨primary keys⟩}`

This defines ⟨*cs*⟩ to be a macro taking one mandatory argument which should contain a ⟨*key*⟩=⟨*value*⟩ list. You can use as many primary keys as you want. The primary keys will be forwarded as a single argument containing every key to ⟨*after*⟩. You can use a single macro for ⟨*after*⟩ or use some arbitrary stuff, which will be left in the input stream before the hashed ⟨*key*⟩=⟨*value*⟩ list with one set of braces stripped. In the macro called in ⟨*after*⟩ you can access the ⟨*value*⟩ of a ⟨*key*⟩ by using `\ekvcValue{⟨key⟩}{#1}` (or whichever argument the hashed ⟨*key*⟩=⟨*value*⟩ list will be).

\ekvcHashAndUse `\ekvcHashAndUse⟨cs⟩{⟨primary keys⟩}`

This will roughly do the same as `\ekvcHashAndForward`, but instead of specifying what will be used after hashing the keys, ⟨*cs*⟩ will use what follows the ⟨*key*⟩=⟨*value*⟩ list. So its syntax will be

⟨*cs*⟩{⟨*key*⟩=⟨*value*⟩, ...}{⟨*after*⟩}

\ekvcValue `\ekvcValue{⟨key⟩}{⟨key list⟩}`

This is a safe (but slow) way to access your keys in a hash variant. ⟨*key*⟩ is the key which's ⟨*value*⟩ you want to use out of the ⟨*key list*⟩. ⟨*key list*⟩ should be the key list argument forwarded to your underlying macro by `\ekvcHash`, `\ekvcHashAndForward`, or `\ekvcHashAndUse`. It will be tested whether the hash function to access that ⟨*key*⟩ exists, the ⟨*key*⟩ argument is not empty, and that the ⟨*key list*⟩ really contains a ⟨*value*⟩ of that ⟨*key*⟩. This macro needs exactly two steps of expansion.

\ekvcValueFast `\ekvcValueFast{⟨key⟩}{⟨key list⟩}`

This behaves just like `\ekvcValue`, but *without any* safety tests. As a result this is about 1.4 times faster *but* will throw low level TEX errors eventually if the hash function isn't defined or the ⟨*key*⟩ isn't part of the ⟨*key list*⟩ (*e.g.*, because it was defined as a key for another macro – all macros share the same hash function per ⟨*key*⟩). Use it if you know what you're doing. This macro needs exactly three steps of expansion in the no-errors case.

\ekvcValueSplit `\ekvcValueSplit{⟨key⟩}{⟨key list⟩}{⟨next⟩}`

If you need a specific ⟨*key*⟩ from a ⟨*key list*⟩ more than once, it'll be a good idea to only extract it once and from then on keep it as a separate argument. Hence the macro `\ekvcValueSplit` will extract one specific ⟨*key*⟩'s value from the list and forward the remainder of the list as the first and the ⟨*key*⟩'s value as the second argument to ⟨*next*⟩, so the result of this will be ⟨*next*⟩{⟨*key list'*⟩}{⟨*value*⟩} with ⟨*key list'*⟩ the remaining list. This is almost as fast as `\ekvcValue` and runs the same tests. Keep in mind that you can't fetch for the same ⟨*key*⟩ again from ⟨*key list'*⟩ as it got removed.

\ekvcValueSplitFast `\ekvcValueSplitFast{⟨key⟩}{⟨key list⟩}{⟨next⟩}`

This behaves just like `\ekvcValueSplit`, but it won't run the same tests, hence it is faster but more error prone, just like the relation between `\ekvcValue` and `\ekvcValueFast`.

## 1.2 Secondary Keys

To remove some of the limitations with the approach that each primary key matches an argument or hash entry, you can define secondary keys. Those have to be defined for each macro but it doesn't matter whether that macro was a split or a hash variant. If a secondary key references another key it doesn't matter whether that other key is primary or secondary.

Secondary keys can have a prefix (like `long`) which are called p-type prefix and must have a type (like `meta`) which are called `t`-type prefix. Some types might require some p-prefixes, while others might forbid those.

Please keep in mind that key names shouldn't start with `EKVC|`.

\ekvcSecondaryKeys `\ekvcSecondaryKeys⟨cs⟩{⟨key⟩=⟨value⟩, ...}`

This is the front facing macro to define secondary keys. For the macro ⟨*cs*⟩ define ⟨*key*⟩ to have definition ⟨*value*⟩. The general syntax for ⟨*key*⟩ should be

⟨*prefix*⟩ ⟨*name*⟩

Where ⟨*prefix*⟩ is a space separated list of optional p-type prefixes followed by one `t`-type prefix. The syntax of ⟨*value*⟩ is dependent on the used `t`-prefix.

### 1.2.1 p-type Prefixes

There is only one p-prefix available, which is `long`.

long The following key will be defined `\long`.

### 1.2.2 t-type Prefixes

If you're familiar with expkv-def you'll notice that the `t`-type prefixes provided here are much fewer. The expansion only concept doesn't allow for great variety in the auto-defined keys.

The syntax examples of the `t`-prefixes will show which p-prefix will be automatically used by printing those black (`long`), which will be available in grey (`long`), and which will be disallowed in red (`long`). This will be put flush right next to the syntax line.

meta `meta ⟨key⟩ = {⟨key⟩=⟨value⟩, ...}` long

With a `meta` key you can set other keys. Whenever ⟨*key*⟩ is used the keys in the ⟨*key*⟩=⟨*value*⟩ list will be set to the values given there. You can use the ⟨*value*⟩ given to ⟨*key*⟩ by using `#1` in the ⟨*key*⟩=⟨*value*⟩ list. The keys in the ⟨*key*⟩=⟨*value*⟩ list can be primary and secondary ones.

nmeta `nmeta ⟨key⟩ = {⟨key⟩=⟨value⟩, ...}` long

An `nmeta` key is like a `meta` key, but it doesn't take a value, so the ⟨*key*⟩=⟨*value*⟩ list is static.

alias `alias ⟨key⟩ = ⟨key2⟩` long

This assigns the definition of ⟨key2⟩ to ⟨key⟩. As a result ⟨key⟩ is an alias for ⟨key2⟩ behaving just the same. Both the value taking and the `NoVal` version (that's expkv slang for a key not accepting a value) will be copied if they are defined when `alias` is used. Of course, ⟨key2⟩ has to be defined, be it as a primary or secondary one.

default `default ⟨key⟩ = {⟨default⟩}` long

If ⟨key⟩ is a defined value taking key, you can define a `NoVal` version with this that will behave as if ⟨key⟩ was given ⟨default⟩ as its ⟨value⟩. Note that this doesn't change the initial values of primary keys set at definition time in `\ekvcSplit` and friends. ⟨key⟩ can be a primary or secondary key.

## 1.3 Example

How could a documentation be a good documentation without some basic examples? Say we want to define a small macro expanding to some character description (who knows why this has to be expandable?). A character description will not have too many items to it, so we use `\ekvcSplit`.

```
\ekvcSplit\character
  {
    name=John Doe,
    age=any,
    nationality=the Universe,
    hobby=to exist,
    type=Mister,
    pronoun=He,
    possessive=his,
  }
  {%
    #1 is a #5 from #3. #6 is of #2 age and #7 hobby is #4.\par
  }
```

Also we want to give some short cuts so that it's easier to describe several persons.

```
\ekvcSecondaryKeys\character
  {
    alias pro = pronoun,
    alias pos = possessive,
    nmeta me =
      {
        name=Jonathan P. Spratte,
        age=a young,
        nationality=Germany,
        hobby=\TeX\ coding,
      },
    meta lady =
      {type=Lady, pronoun=She, possessive=her, name=Jane Doe, #1},
    nmeta paulo =
      {
```

```
      name=Paulo,
      type=duck,
      age=a young,
      nationality=Brazil,
      hobby=to quack,
    }
}
```

Now we can describe people using

```
\character{}
\character{me}
\character{paulo}
\character
  {lady={name=Evelyn,nationality=Ireland,age=the best,hobby=reading}}
\character
  {
    name=Our sun, type=star, nationality=our solar system, pro=It,
    age=an old, pos=its, hobby=shining
  }
```

As one might see, the `lady` key could actually have been an `nmeta` key as well, as all that is done with the argument is using it as a ⟨`key`⟩=⟨`value`⟩ list.

Using xparse and forwarding arguments one can easily define ⟨`key`⟩=⟨`value`⟩ macros with actual optional and mandatory arguments as well. A small nonsense example (which should perhaps use `\ekvcSplitAndForward` instead of `\ekvcHashAndForward` since it only uses four keys and one other argument – and isn't expandable since it uses a `tabular` environment):

```
\usepackage{xparse}
\makeatletter
\NewExpandableDocumentCommand\nonsense{O{} m}{\nonsense@a{#1}{#2}}
\ekvcHashAndForward\nonsense@a\nonsense@b
  {
    keyA = A,
    keyB = B,
    keyC = c,
    keyD = d,
  }
\newcommand*\nonsense@b[2]
  {%
    \begin{tabular}{lll}
      key & A & \ekvcValue{keyA}{#1} \\
          & B & \ekvcValue{keyB}{#1} \\
          & C & \ekvcValue{keyC}{#1} \\
          & D & \ekvcValue{keyD}{#1} \\
      \multicolumn{2}{l}{mandatory} & #2 \\
    \end{tabular}\par
  }
\makeatother
```

And then we would be able to do some nonsense:

```
\nonsense{}
\nonsense[keyA=hihi]{haha}
\nonsense[keyA=hihi, keyB=A]{hehe}
\nonsense[keyC=huhu, keyA=hihi, keyB=A]{haha}
```

## 1.4 Speed Considerations

As already mentioned in the introduction there are some speed considerations implied if you choose to define macros via **expkv|cs**. However the overhead isn't the factor which should hinder you to use **expkv|cs** if you found a reasonable use case. The key-parsing is still faster than with most other ⟨*key*⟩=⟨*value*⟩ packages (see the "Comparisons" subsection in the **expkv** documentation).

The speed considerations in this subsection use the first example in this documentation as the benchmark. So we have seven keys and a short sentence which should be typeset. For comparisons I use the following equivalent **expkv|def** definitions. Each result is the average between changing no keys from their initial values and altering four. Furthermore I'll compare three variants of **expkv|cs** with the **expkv|def** definitions, namely the split example from above, a hash variant using `\ekvcValue` and a hash variant using `\ekvcValueFast`.

```
\usepackage{expkv-def}
\ekvdefinekeys{keys}
  {%
    ,store    name          = \KEYSname
    ,initial  name          = John Doe
    ,store    age           = \KEYSage
    ,initial  age           = any
    ,store    nationality   = \KEYSnationality
    ,initial  nationality   = the Universe
    ,store    hobby         = \KEYShobby
    ,initial  hobby         = to exist
    ,store    type          = \KEYStype
    ,initial  type          = Mister
    ,store    pronoun       = \KEYSpronoun
    ,initial  pronoun       = He
    ,store    possessive    = \KEYSpossessive
    ,initial  possessive    = his
  }
\newcommand*\KEYS[1]
  {%
    \begingroup
      \ekvset{keys}{#1}%
      \KEYSname\ is a \KEYStype\ from \KEYSnationality. \KEYSpronoun\ is
      of \KEYSage\ age and \KEYSpossessive\ hobby is \KEYShobby.%
    \endgroup
  }
```

The first comparison removes the typesetting part from all the definitions, so that only the key parsing is compared. In this comparison the `\ekvcValue` and `\ekvcValueFast` variants will not differ, as they are exactly the same until the key

usage. We find that the split approach is 1.4 times slower than the expkv|def setup and the hash variants end up in the middle at 1.17 times slower.

Next we put the typesetting part back in. Every call of the macros will typeset the sentences into a box register in horizontal mode. With the typesetting part (which includes the accessing of values) the fastest remains the expkv|def definitions, but split is close at 1.16 times slower, followed by the hash variant with fast accesses at 1.36 times slower, and the safe hash access variant ranks in the slowest 1.8 times slower than expkv|def.

Just in case you're wondering now, a simple macro taking seven arguments is 30 to 40 times faster than any of those in the argument grabbing and ⟨*key*⟩=⟨*value*⟩ parsing part and only 1.5 to 2.8 times faster if the typesetting part is factored in. So the real choke isn't the parsing.

So to summarize this, if you have a reasonable use case for expandable ⟨*key*⟩=⟨*value*⟩ parsing macros you should go on and define them using expkv|cs. If you have a reasonable use case for ⟨*key*⟩=⟨*value*⟩ parsing macros but defining them expandable isn't necessary for your use you should take advantage of the greater flexibility of non-expandable ⟨*key*⟩=⟨*value*⟩ setups (but if you're after maximum speed there aren't that many ⟨*key*⟩=⟨*value*⟩ parsers beating expkv|cs). And if you are after maximum performance maybe ditching the ⟨*key*⟩=⟨*value*⟩ interface altogether is a good idea, but depending on the number of arguments your interface might get convoluted.

## 1.5 Useless Macros

Perhaps these macros aren't completely useless, but I figured from a user's point of view I wouldn't know what I should do with these.

`\ekvcDate`
`\ekvcVersion`

These two macros store the version and the date of the package/generic code.

## 1.6 Bugs

Of course I don't think there are any bugs (who would knowingly distribute buggy software as long as he isn't a multi-million dollar corporation?). But if you find some please let me know. For this one might find my email address on the first page or file an issue on Github: https://github.com/Skillmon/tex_expkv-cs

## 1.7 License

# 2 Implementation

## 2.1 The LaTeX Package

Just like for expkv we provide a small LaTeX package that sets up things such that we behave nicely on LaTeX packages and files system. It'll `\input` the generic code which implements the functionality.

```
\RequirePackage{expkv}
\def\ekvc@tmp
  {%
    \ProvidesFile{expkv-cs.tex}%
      [%
        \ekvcDate\space v\ekvcVersion\space
        define expandable key=val macros using expkv%
      ]%
  }
\input{expkv-cs.tex}
\ProvidesPackage{expkv-cs}%
  [%
    \ekvcDate\space v\ekvcVersion\space
    define expandable key=val macros using expkv%
  ]
```

## 2.2 The Generic Code

The rest of this implementation will be the generic code.

Load expkv if the package didn't already do so – since expkv has safeguards against being loaded twice this does no harm and the overhead isn't that big. Also we reuse some of the internals of expkv to save us from retyping them.

```
\input expkv
```

We make sure that `expkv-cs.tex` is only input once:

```
\expandafter\ifx\csname ekvcVersion\endcsname\relax
\else
  \expandafter\endinput
\fi
```

`\ekvcVersion` `\ekvcDate`

We're on our first input, so lets store the version and date in a macro.

```
\def\ekvcVersion{0.4}
\def\ekvcDate{2020-08-08}
```

(*End definition for* `\ekvcVersion` *and* `\ekvcDate`*. These functions are documented on page* 9.)

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

```
\csname ekvc@tmp\endcsname
```

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

```
\expandafter\chardef\csname ekvc@tmp\endcsname=\catcode`\@
\catcode`\@=11
```

`\ekvc@tmp` will be reused later, but we don't need it to ever store information long-term after expkv|cs was initialized.

\ekvc@keycount We'll need to keep count how many keys must be defined for each macro in the `split` variants.

```
\newcount\ekvc@keycount
```

(*End definition for* `\ekvc@keycount`.)

\ekvc@long \ekvc@any@long Some macros will have to be defined long. These two will be let to `\long` when this should be the case.

```
\def\ekvc@long{}
\def\ekvc@any@long{}
```

(*End definition for* `\ekvc@long` *and* `\ekvc@any@long`.)

\ekvc@ekvset@per@expander \ekvc@ekvset@per@expander@a \ekvc@ekvset@per@expander@b This macro expands `\ekvset` twice so that the first two steps of expansion don't have to be made every time the expkv-cs macros are used. We have to do a little magic trick to get the macro parameter `#1` for the macro definition this is used in, even though we're calling `\unexpanded`. We do that by splitting the expanded `\ekvset` at some marks and place `##1` in between.

```
\def\ekvc@ekvset@pre@expander#1%
  {%
    \expandafter\ekvc@ekvset@pre@expander@a\ekvset{#1}\ekvc@stop\ekvc@stop
  }
\def\ekvc@ekvset@pre@expander@a
  {%
    \expandafter\ekvc@ekvset@pre@expander@b
  }
\def\ekvc@ekvset@pre@expander@b#1\ekvc@stop#2\ekvc@stop
  {%
    \unexpanded{#1}##1\unexpanded{#2}%
  }
```

(*End definition for* `\ekvc@ekvset@per@expander`, `\ekvc@ekvset@per@expander@a`, *and* `\ekvc@ekvset@per@expander@b`.)

\ekvcSplitAndUse The first user macro we want to set up can be reused for `\ekvcSplitAndForward` and `\ekvcSplit`. We'll split this one up so that the test whether the macro is already defined doesn't run twice.

```
\protected\long\def\ekvcSplitAndUse#1#2%
  {%
    \ekv@ifdefined{\expandafter\@gobble\string#1}%
      {\ekvc@err@already@defined#1}%
      {\ekvcSplitAndUse@#1{}{#2}}%
  }
```

(*End definition for* `\ekvcSplitAndUse`. *This function is documented on page* 3.)

\ekvcSplitAndUse@ The actual macro setting up things. We need to set some variables, forward the key list to `\ekvc@SetupSplitKeys`, and afterwards define the front facing macro to call `\ekvset` and put the initials and the argument sorting macro behind it. The internals `\ekvc@any@long`, `\ekvc@initials` and `\ekvc@keycount` will be set correctly by `\ekvc@SetupSplitKeys`.

```
\protected\long\def\ekvcSplitAndUse@#1#2#3%
  {%
    \edef\ekvc@set{\string#1}%
    \ekvc@SetupSplitKeys{#3}%
```

```
    \ekvc@any@long\edef#1##1%
      {%
        \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}%
        \unexpanded\expandafter
          {\csname ekvc@split@\the\ekvc@keycount\endcsname}%
        \unexpanded\expandafter{\ekvc@initials{}#2}%
      }%
  }
```

(*End definition for* `\ekvcSplitAndUse@`.)

`\ekvcSplitAndForward` This just reuses `\ekvcSplitAndUse@` with a non-empty second argument, resulting in that argument to be called after the splitting.

```
\protected\long\def\ekvcSplitAndForward#1#2#3%
  {%
    \ekv@ifdefined{\expandafter\@gobble\string#1}%
      {\ekvc@err@already@defined#1}%
      {\ekvcSplitAndUse@#1{{#2}}{#3}}%
  }
```

(*End definition for* `\ekvcSplitAndForward`. *This function is documented on page* 3.)

`\ekvcSplit` The first half is just `\ekvcSplitAndForward` then we define the macro to which the parsed key list is forwarded. There we need to allow for up to nine arguments.

```
\protected\long\def\ekvcSplit#1#2#3%
  {%
    \ekv@ifdefined{\expandafter\@gobble\string#1}%
      {\ekvc@err@already@defined#1}%
      {%
        \expandafter
        \ekvcSplitAndUse@\expandafter#1\csname ekvc@\string#1\endcsname{#2}%
        \ifnum\ekvc@keycount=0
          \def\ekvc@tmp##1##{}%
        \else
          \ifnum\ekvc@keycount>9
            \ekvc@err@toomany{#1}%
            \ekvc@defarggobbler9%
          \else
            \expandafter\ekvc@defarggobbler\the\ekvc@keycount
          \fi
        \fi
        \ekvc@any@long\expandafter
        \def\csname ekvc@\string#1\expandafter\endcsname
            \ekvc@tmp##1##2##3##4##5##6##7##8##9%
          {#3}%
      }%
  }
```

(*End definition for* `\ekvcSplit`. *This function is documented on page* 3.)

`\ekvc@SetupSplitKeys` `\ekvc@SetupSplitKeys@a` `\ekvc@SetupSplitKeys@b` `\ekvc@SetupSplitKeys@c` These macros parse the list of keys and set up the key macros. First we need to initialise some macros and start `\ekvparse`.

```
\protected\long\def\ekvc@SetupSplitKeys
  {%
```

```
    \ekvc@keycount=0
    \def\ekvc@any@long{}%
    \def\ekvc@initials{}%
    \ekvparse\ekvc@err@value@required\ekvc@SetupSplitKeys@a
  }
```

Then we need to step the key counter for each key. Also we have to check whether this key has a `long` prefix so we initialise `\ekvc@long`.

```
\protected\def\ekvc@SetupSplitKeys@a#1%
  {%
    \advance\ekvc@keycount1
    \def\ekvc@long{}%
    \ekvc@ifspace{#1}%
      {\ekvc@SetupSplitKeys@b#1\ekvc@stop}%
      {\ekvc@SetupSplitKeys@c{#1}}%
  }
```

If there was a space, there might be a prefix. If so call the prefix macro, else call the next step `\ekvc@SetupSplitKeys@c` which will define the key macro and add the key's value to the initials list.

```
\protected\def\ekvc@SetupSplitKeys@b#1 #2\ekvc@stop
  {%
    \ekv@ifdefined{ekvc@split@p@#1}%
      {\csname ekvc@split@p@#1\endcsname{#2}}%
      {\ekvc@SetupSplitKeys@c{#1 #2}}%
  }
```

The inner definition is grouped, because we don't want to actually define the marks we build with `\csname`. We have to append the value to the `\ekvc@initials` list here with the correct split mark. The key macro will read everything up to those split marks and change the value following it to the value given to the key. Additionally we'll need a sorting macro for each key count in use so we set it up with `\ekvc@setup@splitmacro`.

```
\protected\long\def\ekvc@SetupSplitKeys@c#1#2%
  {%
    \begingroup
    \edef\ekvc@tmp
      {%
        \endgroup
        \long\def\unexpanded{\ekvc@tmp}####1####2%
            \unexpanded\expandafter
            {\csname ekvc@splitmark@\the\ekvc@keycount\endcsname}####3%
          {%
            ####2%
            \unexpanded\expandafter
              {\csname ekvc@splitmark@\the\ekvc@keycount\endcsname}{####1}%
          }%
```

The short variant needs a bit of special treatment. The key macro will be short to throw the correct error, but since there might be long macros somewhere the reordering of arguments needs to be long, so for short keys we use a two step approach, first grabbing only the short argument, then reordering.

```
        \unless\ifx\ekvc@long\long
          \let\unexpanded\expandafter
            {\csname ekvc@\ekvc@set(#1)\endcsname\ekvc@tmp}%
          \def\unexpanded{\ekvc@tmp}####1%
```

```
            {%
              \unexpanded\expandafter{\csname ekvc@\ekvc@set(#1)\endcsname}%
                {####1}%
            }%
        \fi
        \def\unexpanded{\ekvc@initials}%
          {%
            \unexpanded\expandafter{\ekvc@initials}%
            \unexpanded\expandafter
              {\csname ekvc@splitmark@\the\ekvc@keycount\endcsname{#2}}%
          }%
      }%
    \ekvc@tmp
    \ekvlet\ekvc@set{#1}\ekvc@tmp
    \expandafter\ekvc@setup@splitmacro\expandafter{\the\ekvc@keycount}%
  }
```

(*End definition for* `\ekvc@SetupSplitKeys` *and others.*)

`\ekvc@split@p@long` The `long` prefix lets the internals `\ekvc@long` and `\ekvc@any@long` to `\long` so that the key macro will be long.

```
\protected\def\ekvc@split@p@long
  {%
    \let\ekvc@long\long
    \let\ekvc@any@long\long
    \ekvc@SetupSplitKeys@c
  }
```

(*End definition for* `\ekvc@split@p@long`.)

`\ekvc@defarggobbler` This is needed to define a macro with 1-9 parameters programmatically. LaTeX's `\newcommand` does something similar for example.

```
\protected\def\ekvc@defarggobbler#1{\def\ekvc@tmp##1#1##2##{##1#1}}
```

(*End definition for* `\ekvc@defarggobbler`.)

`\ekvc@setup@splitmacro` `\ekvc@split@1` Since the first split macro is different from the others we manually set that one up now. All the others will be defined as needed (always globally). The split macros just read up until the correct split mark, move that argument into a list and reinsert the rest, calling the next split macro afterwards.

```
\begingroup
\edef\ekvc@tmp
  {%
    \long\gdef\unexpanded\expandafter{\csname ekvc@split@1\endcsname}%
        \unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}%
        ##1##2##3%
      {##3{##1}##2}%
  }
\ekvc@tmp
\endgroup
\protected\def\ekvc@setup@splitmacro#1%
  {%
    \ekv@ifdefined{ekvc@split@#1}{}%
      {%
```

```
      \begingroup
        \edef\ekvc@tmp
          {%
            \long\gdef
              \unexpanded\expandafter{\csname ekvc@split@#1\endcsname}%
              ####1%
              \unexpanded\expandafter{\csname ekvc@splitmark@#1\endcsname}%
              ####2####3%
            {%
              \unexpanded\expandafter
                {\csname ekvc@split@\the\numexpr#1-1\relax\endcsname}%
              ####1{{####2}####3}%
            }%
          }%
        \ekvc@tmp
      \endgroup
    }%
  }
```

(*End definition for* `\ekvc@setup@splitmacro` *and* `\ekvc@split@1`.)

`\ekvcHashAndUse` `\ekvcHashAndUse` works just like `\ekvcSplitAndUse`.

```
\protected\long\def\ekvcHashAndUse#1#2%
  {%
    \ekv@ifdefined{\expandafter\@gobble\string#1}%
      {\ekvc@err@already@defined#1}%
      {\ekvcHashAndUse@#1{}{#2}}%
  }
```

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

`\ekvcHashAndForward@` This is more or less the same as `\ekvcSplitAndUse@`. Instead of an empty group we place a marker after the initials, we don't use the sorting macros of `split`, but instead pack all the values in one argument.

```
\protected\long\def\ekvcHashAndUse@#1#2#3%
  {%
    \edef\ekvc@set{\string#1}%
    \ekvc@SetupHashKeys{#3}%
    \ekvc@any@long\edef#1##1%
      {%
        \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}%
        \unexpanded{\ekvc@hash@pack@argument}%
        \unexpanded\expandafter{\ekvc@initials\ekvc@stop#2}%
      }%
  }
```

(*End definition for* `\ekvcHashAndForward@`.)

`\ekvcHashAndForward` `\ekvcHashAndForward` works just like `\ekvcSplitAndForward`.

```
\protected\long\def\ekvcHashAndForward#1#2#3%
  {%
    \ekv@ifdefined{\expandafter\@gobble\string#1}%
      {\ekvc@err@already@defined#1}%
      {\ekvcHashAndUse@#1{{#2}}{#3}}%
  }
```

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

`\ekvcHash` `\ekvcHash` does the same as `\ekvcSplit`, but has the advantage of not needing to count arguments, so the definition of the internal macro is a bit more straight forward.

```
\protected\long\def\ekvcHash#1#2#3%
  {%
    \ekv@ifdefined{\expandafter\@gobble\string#1}%
      {\ekvc@err@already@defined#1}%
      {%
        \expandafter
        \ekvcHashAndUse@\expandafter#1\csname ekvc@\string#1\endcsname{#2}%
        \ekvc@any@long\expandafter\def\csname ekvc@\string#1\endcsname##1{#3}%
      }%
  }
```

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

`\ekvc@hash@pack@argument` All this macro does is pack the values into one argument and forward that to the next macro.

```
\long\def\ekvc@hash@pack@argument#1\ekvc@stop#2{#2{#1}}
```

(*End definition for* `\ekvc@hash@pack@argument`.)

`\ekvc@SetupHashKeys` `\ekvc@SetupHashKeys@a` `\ekvc@SetupHashKeys@b` This should look awfully familiar as well, since it's just the same as for the split keys with a few other names here and there.

```
\protected\long\def\ekvc@SetupHashKeys#1%
  {%
    \def\ekvc@any@long{}%
    \def\ekvc@initials{}%
    \ekvparse\ekvc@err@value@required\ekvc@SetupHashKeys@a{#1}%
  }
\protected\def\ekvc@SetupHashKeys@a#1%
  {%
    \def\ekvc@long{}%
    \ekvc@ifspace{#1}%
      {\ekvc@SetupHashKeys@b#1\ekvc@stop}%
      {\ekvc@SetupHashKeys@c{#1}}%
  }
\protected\def\ekvc@SetupHashKeys@b#1 #2\ekvc@stop
  {%
    \ekv@ifdefined{ekvc@hash@p@#1}%
      {\csname ekvc@hash@p@#1\endcsname{#2}}%
      {\ekvc@SetupHashKeys@c{#1 #2}}%
  }
```

Yes, even the defining macro looks awfully familiar. Instead of numbered we have named marks. Still the key macros grab everything up to their respective mark and reorder the arguments. The same quirk is applied for short keys. And instead of the `\ekvc@setup@splitmacro` we use `\ekvc@setup@hashmacro`.

```
\protected\long\def\ekvc@SetupHashKeys@c#1#2%
  {%
    \begingroup
    \edef\ekvc@tmp
      {%
```

```
        \endgroup
        \long\def\unexpanded{\ekvc@tmp}####1####2%
            \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}####3%
          {%
            ####2%
            \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}{####1}%
          }%
        \unless\ifx\ekvc@long\long
          \let\unexpanded\expandafter
            {\csname ekvc@\ekvc@set(#1)\endcsname\ekvc@tmp}%
          \def\unexpanded{\ekvc@tmp}####1%
            {%
              \unexpanded\expandafter{\csname ekvc@\ekvc@set(#1)\endcsname}%
                {####1}%
            }%
        \fi
        \def\unexpanded{\ekvc@initials}%
          {%
            \unexpanded\expandafter{\ekvc@initials}%
            \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname{#2}}%
          }%
      }%
    \ekvc@tmp
    \ekvlet\ekvc@set{#1}\ekvc@tmp
    \ekvc@setup@hashmacro{#1}%
  }
```

(*End definition for* `\ekvc@SetupHashKeys`, `\ekvc@SetupHashKeys@a`, *and* `\ekvc@SetupHashKeys@b`.)

`\ekvc@hash@p@long` Nothing astonishing here either.

```
\protected\def\ekvc@hash@p@long
  {%
    \let\ekvc@long\long
    \let\ekvc@any@long\long
    \ekvc@SetupHashKeys@c
  }
```

(*End definition for* `\ekvc@hash@p@long`.)

`\ekvc@setup@hashmacro` The safe hash macros will be executed inside of a `\romannumeral` expansion context, so they have to insert a stop mark for that once they are done. Most of the tests which have to be executed will already be done, but we have to play safe if the hash doesn't show up in the hash list. Therefore we use some `\ekvc@marks` and `\ekvc@stop` to throw errors if the hash isn't found in the right place. The fast variants have an easier life and just return the correct value.

```
\protected\def\ekvc@setup@hashmacro#1%
  {%
    \ekv@ifdefined{ekvc@hash@#1}{}%
      {%
        \begingroup
          \edef\ekvc@tmp
            {%
              \long\gdef
                \unexpanded\expandafter{\csname ekvc@fasthash@#1\endcsname}%
```

```
      ####1%
      \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}%
      ####2####3\unexpanded{\ekvc@stop}%
    {####2}%
  \long\gdef
    \unexpanded\expandafter{\csname ekvc@safehash@#1\endcsname}%
      ####1%
    {%
      \unexpanded\expandafter{\csname ekvc@@safehash@#1\endcsname}%
      ####1\unexpanded{\ekvc@mark}{ }%
      \unexpanded\expandafter
        {%
          \csname ekvc@hashmark@#1\endcsname
          {\ekvc@err@missing@hash{#1} }%
          \ekvc@mark{}\ekvc@stop
        }%
    }%
  \long\gdef
    \unexpanded\expandafter{\csname ekvc@@safehash@#1\endcsname}%
      ####1%
      \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}%
      ####2####3\unexpanded{\ekvc@mark}####4####5%
      \unexpanded{\ekvc@stop}%
    {%
      ####4####2%
    }%
  \long\gdef\unexpanded\expandafter
    {\csname ekvc@fastsplithash@#1\endcsname}%
      ####1%
      \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}%
      ####2####3\unexpanded{\ekvc@stop}####4%
    {%
      ####4{####1####3}{####2}%
    }%
  \long\gdef\unexpanded\expandafter
    {\csname ekvc@safesplithash@#1\endcsname}####1%
    {%
      \unexpanded\expandafter
        {\csname ekvc@@safesplithash@#1\endcsname}%
      ####1\unexpanded{\ekvc@mark\ekvc@safe@found@hash}%
      \unexpanded\expandafter
        {%
          \csname ekvc@hashmark@#1\endcsname{}%
          \ekvc@mark{\ekvc@err@missing@hash{#1}\ekvc@safe@no@hash}%
          \ekvc@stop
        }%
    }%
  \long\gdef\unexpanded\expandafter
    {\csname ekvc@@safesplithash@#1\endcsname}%
      ####1%
      \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}%
      ####2####3\unexpanded{\ekvc@mark}####4####5%
      \unexpanded{\ekvc@stop}%
    {%
```

```
                  ####4{####2}####1####3\unexpanded{\ekvc@stop}%
                }%
              }%
            \ekvc@tmp
          \endgroup
        }%
    }
```

(*End definition for* `\ekvc@setup@hashmacro`.)

`\ekvcValue` All this does is a few consistency checks on the first argument (not empty, hash macro exists) and then call that hash-grabbing macro that will also test whether the hash is inside of #2 or not.

```
\long\def\ekvcValue#1#2%
  {%
    \romannumeral`\^^@%
    \ekv@ifdefined{ekvc@safehash@#1}%
      {\csname ekvc@safehash@#1\endcsname{#2}}%
      {\ekvc@err@unknown@hash{#1} }% keep this space
  }
```

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

`\ekvcValueFast` To be as fast as possible, this doesn't test for anything, assuming the user knows best.

```
\long\def\ekvcValueFast#1#2{\csname ekvc@fasthash@#1\endcsname#2\ekvc@stop}
```

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

`\ekvcValueSplit` This splits off a single value.

```
\long\def\ekvcValueSplit#1#2#3%
  {%
    \ekv@ifdefined{ekvc@safesplithash@#1}%
      {\csname ekvc@safesplithash@#1\endcsname{#2}{#3}}%
      {\ekvc@err@unknown@hash{#1}#3{}{#2}}%
  }
```

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

`\ekvc@safe@found@hash`
`\ekvc@safe@no@hash`

```
\long\def\ekvc@safe@found@hash#1#2\ekvc@stop#3%
  {%
    #3{#2}{#1}%
  }
\long\def\ekvc@safe@no@hash#1#2\ekvc@mark\ekvc@safe@found@hash\ekvc@stop#3%
  {%
    #3{#2}{}%
  }
```

(*End definition for* `\ekvc@safe@found@hash` *and* `\ekvc@safe@no@hash`.)

`\ekvcValueSplitFast` Again a fast approach which doesn't provide too many safety measurements. This needs to build the hash function and expand it before passing the results to the next control sequence. The first step only builds the control sequence.

```
\long\def\ekvcValueSplitFast#1#2%
  {%
    \csname ekvc@fastsplithash@#1\endcsname#2\ekvc@stop
  }
```

(*End definition for* `\ekvcValueSplitFast`. *This function is documented on page* 5.)

`\ekvcValueSplitFast@a` This step then expands the hash function once and passes the result to #3 which should be a single control sequence.

```
\long\def\ekvcValueSplitFast@#1#2#3%
  {%
    \expandafter#3\expandafter{#1#2\ekvc@stop}%
  }
```

(*End definition for* `\ekvcValueSplitFast@a`.)

`\ekvc@safehash@` `\ekvc@fasthash@` `\ekvc@safesplithash@` `\ekvc@fastsplithash@` At least in the empty hash case we can provide a meaningful error message without affecting performance by just defining the macro that would be build in that case. There is of course a downside to this, the error will not be thrown by `\ekvcValueFast` in three expansion steps. The safe hash variant has to also stop the `\romannumeral` expansion.

```
\long\def\ekvc@safehash@#1{\ekvc@err@empty@hash\@gobble{} }% keep this space
\long\def\ekvc@fasthash@#1\ekvc@stop{\ekvc@err@empty@hash}
\long\def\ekvc@safesplithash@#1#2{\ekvc@err@empty@hash#2{#1}{}}
\long\def\ekvc@fastsplithash@#1\ekvc@stop#2{\ekvc@err@empty@hash#2{#1}{}}
```

(*End definition for* `\ekvc@safehash@` *and others.*)

`\ekvcSecondaryKeys` The secondary keys are defined pretty similar to the way the originals are, but here we also introduce some key types (those have a `@t@` in their name) additionally to the prefixes.

```
\protected\long\def\ekvcSecondaryKeys#1#2%
  {%
    \edef\ekvc@set{\string#1}%
    \ekvparse\ekvc@err@value@required\ekvcSecondaryKeys@a{#2}%
  }
\protected\def\ekvcSecondaryKeys@a#1%
  {%
    \def\ekvc@long{}%
    \ekvc@ifspace{#1}%
      {\ekvcSecondaryKeys@b#1\ekvc@stop}%
      {\ekvc@err@missing@type{#1}\@gobble}%
  }
\protected\def\ekvcSecondaryKeys@b#1 #2\ekvc@stop
  {%
    \ekv@ifdefined{ekvc@p@#1}%
      {\csname ekvc@p@#1\endcsname}%
      {%
        \ekv@ifdefined{ekvc@t@#1}%
          {\csname ekvc@t@#1\endcsname}%
          {\ekvc@err@unknown@keytype{#1}\@firstoftwo\@gobble}%
      }%
      {#2}%
  }
```

(*End definition for* `\ekvcSecondaryKeys`. *This function is documented on page* 5.)

### 2.2.1 Secondary Key Types

\ekvc@p@long
\ekvc@after@ptype

The prefixes are pretty straight forward again. Just set \ekvc@long and forward to the @t@ type.

```
\protected\def\ekvc@p@long#1%
  {%
    \ekvc@ifspace{#1}%
      {%
        \let\ekvc@long\long
        \ekvc@after@ptype#1\ekvc@stop
      }%
      {\ekvc@err@missing@type{long #1}\@gobble}%
  }
\protected\def\ekvc@after@ptype#1 #2\ekvc@stop
  {%
    \ekv@ifdefined{ekvc@t@#1}%
      {\csname ekvc@t@#1\endcsname{#2}}%
      {\ekvc@err@unknown@keytype{#1}\@gobble}%
  }
```

(*End definition for* \ekvc@p@long *and* \ekvc@after@ptype.)

\ekvc@t@meta
\ekvc@t@nmeta
\ekvc@type@meta
\ekvc@type@meta@a
\ekvc@type@meta@b

The meta and nmeta key types use a nested \ekvset to set other keys in the same macro's ⟨*set*⟩.

```
\protected\def\ekvc@t@meta
  {%
    \edef\ekvc@tmp{\ekvc@set}%
    \expandafter\ekvc@type@meta\expandafter{\ekvc@tmp}\ekvc@long{##1}\ekvlet
  }
\protected\def\ekvc@t@nmeta#1%
  {%
    \ekvc@assert@not@long{nmeta #1}%
    \edef\ekvc@tmp{\ekvc@set}%
    \expandafter\ekvc@type@meta\expandafter{\ekvc@tmp}{}{}\ekvletNoVal{#1}%
  }
\protected\long\def\ekvc@type@meta#1#2#3#4#5#6%
  {%
    \expandafter\ekvc@type@meta@a\expandafter{\ekvset{#1}{#6}}{#2}{#3}%
    #4\ekvc@set{#5}\ekvc@tmp
  }
\protected\def\ekvc@type@meta@a
  {%
    \expandafter\ekvc@type@meta@b\expandafter
  }
\protected\long\def\ekvc@type@meta@b#1#2#3%
  {%
    #2\def\ekvc@tmp#3{#1}%
  }
```

(*End definition for* \ekvc@t@meta *and others.*)

\ekvc@t@alias

alias just checks whether there is a key and/or NoVal key defined with the target name and \let the key to those.

```
\protected\def\ekvc@t@alias#1#2%
```

```
  {%
    \ekvc@assert@not@long{alias #1}%
    \let\ekvc@tmp\@firstofone
    \ekvifdefined\ekvc@set{#2}%
      {%
        \ekvletkv\ekvc@set{#1}\ekvc@set{#2}%
        \let\ekvc@tmp\@gobble
      }%
      {}%
    \ekvifdefinedNoVal\ekvc@set{#2}%
      {%
        \ekvletkvNoVal\ekvc@set{#1}\ekvc@set{#2}%
        \let\ekvc@tmp\@gobble
      }%
      {}%
    \ekvc@tmp{\ekvc@err@unknown@key{#2}}%
  }
```

(*End definition for* `\ekvc@t@alias`.)

`\ekvc@t@default` The `default` key can be used to set a `NoVal` key for an existing key. It will just pass the ⟨*value*⟩ to the key macro of that other key.

```
\protected\long\def\ekvc@t@default#1#2%
  {%
    \ekvifdefined\ekvc@set{#1}%
      {%
        \ekvc@assert@not@long{default #1}%
        \edef\ekvc@tmp
          {%
            \unexpanded\expandafter
              {\csname\ekv@name\ekvc@set{#1}\endcsname{#2}}%
          }%
        \ekvletNoVal\ekvc@set{#1}\ekvc@tmp
      }%
      {\ekvc@err@unknown@key{#1}}%
  }
```

(*End definition for* `\ekvc@t@default`.)

### 2.2.2 Helper Macros

`\ekvc@ifspace` `\ekvc@ifspace@` A test which can be reduced to an if-empty by gobbling everything up to the first space.

```
\long\def\ekvc@ifspace#1%
  {%
    \ekvc@ifspace@#1 \ekv@ifempty@B
      \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
  }
\long\def\ekvc@ifspace@#1 % keep this space
  {%
    \ekv@ifempty@\ekv@ifempty@A
  }
```

(*End definition for* `\ekvc@ifspace` *and* `\ekvc@ifspace@`.)

### 2.2.3 Assertions

\ekvc@assert@not@long

Some keys don't want to be `long` and we have to educate the user, so let's throw an error if someone wanted these to be long.

```
\long\def\ekvc@assert@not@long#1{\ifx\ekvc@long\long\ekvc@err@no@long{#1}\fi}
```

(*End definition for* `\ekvc@assert@not@long`.)

### 2.2.4 Messages

\ekvc@err@toomany
\ekvc@err@value@required
\ekvc@err@missing@type
\ekvc@err@already@defined

Boring unexpandable error messages.

```
\protected\def\ekvc@err@toomany#1%
  {%
    \errmessage{expkv-cs Error: Too many keys for macro `\string#1'}%
  }
\protected\def\ekvc@err@value@required#1%
  {%
    \errmessage{expkv-cs Error: Missing value for key `\unexpanded{#1}'}%
  }
\protected\def\ekvc@err@missing@type#1%
  {%
    \errmessage
      {expkv-cs Error: Missing type for secondary key `\unexpanded{#1}'}%
  }
\protected\def\ekvc@err@no@long#1%
  {%
    \errmessage
      {expkv-cs Error: prefix `long' not accepted for `\unexpanded{#1}'}%
  }
\protected\def\ekvc@err@already@defined#1%
  {%
    \errmessage{expkv-cs Error: Macro `\string#1' already defined}%
  }
\protected\def\ekvc@err@unknown@keytype#1%
  {%
    \errmessage{expkv-cs Error: Unknown key type `\unexpanded{#1}'}%
  }
\protected\def\ekvc@err@unknown@key#1%
  {%
    \errmessage
      {expkv-cs Error: Unknown key `\unexpanded{#1}' for macro `\ekvc@set'}%
  }
```

(*End definition for* `\ekvc@err@toomany` *and others.*)

\ekvc@err
\ekvc@err@

We need a way to throw error messages expandably in some contexts.

```
\begingroup
\edef\ekvc@err
  {%
    \endgroup
    \unexpanded{\long\def\ekvc@err}##1%
      {%
        \unexpanded{\expandafter\ekvc@err@\@firstofone}%
        {\unexpanded\expandafter{\csname ! expkv-cs Error:\endcsname}##1.}%
```

```
          \unexpanded{\ekv@stop}%
        }%
    }
\ekvc@err
\def\ekvc@err@{\expandafter\ekv@gobbleto@stop}
```

(*End definition for* `\ekvc@err` *and* `\ekvc@err@`.)

`\ekvc@err@unknown@hash` `\ekvc@err@empty@hash` `\ekvc@err@missing@hash`

And here are the expandable error messages.

```
\long\def\ekvc@err@unknown@hash#1{\ekvc@err{unknown hash `#1'}}
\long\def\ekvc@err@missing@hash#1{\ekvc@err{hash `#1' not found}}
\long\def\ekvc@err@empty@hash{\ekvc@err{empty hash}}
```

(*End definition for* `\ekvc@err@unknown@hash`, `\ekvc@err@empty@hash`, *and* `\ekvc@err@missing@hash`.)

Now everything that's left is to reset the category code of `@`.

```
\catcode`\@=\ekvc@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.