expkv|cs

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

Jonathan P. Spratte*

2022-02-13 v1.2

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) consider not using expkv-cs. The interface has some overhead (though it still is fast – check subsection 1.8) 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. Nevertheless setting up macros with expkv-cs, especially with `\ekvcSplit`, is very convenient in my opinion, so if you just want to define a single macro with just a few keys this might be the way to go.

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

```
\input expkv-cs          % plainTeX
\usepackage{expkv-cs}    % LaTeX
\usemodule[expkv-cs]     % ConTeXt
```

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 automatically 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 an explicit `\par` token. Note however that `long` keys are a microscopic grain faster (due to some internals of expkvics). 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`.

To allow keys not defined `long` which key names should start with `long`, you can also use the prefix `short` (`short` and `long` are mutually exclusive, which ever comes first defines the behaviour, the latter is considered part of the key name). Note that this is the only reason for `short`'s existance, essentially it does nothing.

The consequence culminates in the following:

```
\ekvcSplit\foo
  {
     long short = abc\par
    ,short long = def
  }
  {#1#2}
```

will define a macro `\foo` that knows two primary keys, the first one is called `short`, and accepts explicit `\par` tokens inside its values, the second one is called `long` and will not accept `\par` tokens (leading to a low level TEX error). A description of `\ekvcSplit` follows shortly.

There is one exception to these syntax rules of ⟨*primary keys*⟩: One can include a key named `...` without a value. If this is found the ⟨*cs*⟩ will be defined `\long`. Any unknown keys found at use time will end up in a list at this spot. See some examples in subsection 1.4.

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.

Example: The following defines a macro `\foo` that takes the keys `a` and `b` and outputs their values in a textual form:

```
\ekvcSplit\foo{a=a,b=b}{a is #1.\par b is #2.\par}
\foo{}
\foo{b=e}
```

a is a.
b is b.
a is a.
b is e.

\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}
```

In the first case `\foo@aux` should take at least two arguments (`keyA` and `keyB`), in the second case at least three (`more args`, `keyA`, and `keyB`).

\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*⟩}

and the code in `after` should expect at least as many arguments as the number of keys defined for ⟨*cs*⟩.

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 slower (for a rudimentary 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 consider using 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}`.

Example: This defines an equivalent macro to the `\foo` defined with `\ekvcSplit` earlier:

```
\ekvcHash\foo{a=a,b=b}{a is \ekvcValue{a}{#1}.\par
                       b is \ekvcValue{b}{#1}.\par}
\foo{}
\foo{b=e}
```

```
a is a.
b is b.
a is a.
b is e.
```

`\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).

Example: This defines a macro `\foo` processing two keys, and passing the result to `\foobar`:

```
\ekvcHashAndForward\foo\foobar{a=a,b=b}
\newcommand*\foobar[1]{a is \ekvcValue{a}{#1}.\par
                       b is \ekvcValue{b}{#1}.\par}
```

`\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 at install time, ⟨*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 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 and if used inside of an `\edef` or `\expanded` context will protect the ⟨*value*⟩ from further expanding.

`\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*⟩ name). 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 it as an argument to ⟨*next*⟩, so the result of this will be ⟨*next*⟩{⟨*value*⟩}. This is roughly as fast as `\ekvcValue` and runs the same tests.

Example: The following defines a macro `\foo` which will take three keys. Since the next parsing step will need the value of one of the keys multiple times we split that key off the list (in this example the next step doesn't use the key multiple times for simplicity though), and the entire list is forwarded as the second argument:

```
\ekvcHash\foo{a=a,b=b,c=c}
  {\ekvcValueSplit{a}{#1}\foobar{#1}}
\newcommand*\foobar[2]{a is #1.\par
                       b is \ekvcValue{b}{#2}.\par
                       c is \ekvcValue{c}{#2}.\par}
\foo{}
```

a is a.
b is b.
c is c.

\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 individually 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 unless otherwise specified.

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 (see \ekvcChange in subsection 1.3 for this). ⟨key⟩ can be a primary or secondary key.

---

enum `enum ⟨key⟩ = {⟨key₂⟩}{⟨value⟩, ...}` long

This defines ⟨key⟩ to only accept the values given in the list of the second argument of its definition. It forwards the position of ⟨value⟩ in that list to ⟨key₂⟩ (zero-based). The ⟨key₂⟩ has to already be defined by the time an enum key is set up. Each ⟨value⟩ in the list (and at use time) is treated as a string, so no expansion takes place here.

If you use enum twice on the same ⟨key⟩ the new values will again start at zero (so it is possible to define multiple values with the same outcome), however since you can't skip values you'll have to use the same as in the first call for values with just a single variant. There is no interface to delete existing values.

Example: First a small example that might give you an idea of what the description above could mean:

```
\ekvcSplit\foo{k-internal=-1}{#1}
\ekvcSecondaryKeys\foo
  {enum k = {k-internal}{a,b,c}}
\foo{}\foo{k=a}\foo{k=b}\foo{k=c}
```

```
-1012
```

Example: We can define a choice setup that might do different things based on the choice encountered, and the numeric value is easy to parse using `\ifcase`:

```
\ekvcSplit\foo{k-internal=-1}
  {%
    \ifcase#1
      is\or
      This\or
      easy%
    \else
      .%
    \fi
  }
\ekvcSecondaryKeys\foo
  {enum k = {k-internal}{a,b,c}}
\foo{k=b} \foo{k=a} \foo{k=c}\foo{}
```

This is easy.

aggregate

`aggregate` ⟨*key*⟩ `= {`⟨*primary*⟩`}{`⟨*definition*⟩`}` long

While all other key types replace the current value of the associated ⟨*primary*⟩ key, with `aggregate` you can create keys that append or prepend (or whatever you like) the new value to the current one. The value must be exactly two TEX arguments, where ⟨*primary*⟩ should be the name of a ⟨*primary*⟩ key, and ⟨*definition*⟩ the way you want to store the current and the new value. Inside ⟨*definition*⟩ you can use `#1` for the current, and `#2` for the new value. The ⟨*definition*⟩ will not expand any further during the entire parsing (so this doesn't allow general processing of current and new values). The resulting ⟨*key*⟩ will inherit being either `short` or `long` from the ⟨*primary*⟩ key.

Example: The following defines an internal key (`k-internal`), which is used to build a comma separated list from each call of the user facing key (`k`):

```
\ekvcSplit\foo
  {k-internal=0,color=red}
  {\textcolor{#2}{#1}}
\ekvcSecondaryKeys\foo
  {aggregate k = {k-internal}{#1,#2}}
\foo{}\par
\foo{k=1,k=2,k=3,k=4}
```

0
0,1,2,3,4

But also more strange stuff could end there, like macros or using the same value multiple times:

```
\ekvcSecondaryKeys\foo
  {aggregate k = {k-internal}{\old{#1}\new{#2\old{#1}}}}
```

flag-bool

`flag-bool` ⟨*key*⟩ `=` ⟨*cs*⟩ long

This is a secondary key that doesn't directly involve any of the primary or secondary keys. This defines ⟨*key*⟩ to take a value, which should be either `true` or `false`, and set the flag called ⟨*cs*⟩ accordingly as a boolean. If ⟨*cs*⟩ isn't defined yet it will be initialised as a flag. Please also read subsection 1.5.

flag-true
flag-false

`flag-true` ⟨*key*⟩ = ⟨*cs*⟩ long

This is a secondary key that doesn't directly involve any of the primary or secondary keys. This defines ⟨*key*⟩ to take no value and set the flag called ⟨*cs*⟩ to `true` or `false`, respectively. If ⟨*cs*⟩ isn't defined yet it will be initialised as a flag. Please also read subsection 1.5.

flag-raise

`flag-raise` ⟨*key*⟩ = ⟨*cs*⟩ long

This is a secondary key that doesn't directly involve any of the primary or secondary keys. This defines ⟨*key*⟩ to take no value and raise the flag called ⟨*cs*⟩. If ⟨*cs*⟩ isn't defined yet it will be initialised as a flag. Please also read subsection 1.5.

1.3 Changing the Initial Values

\ekvcChange

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

This processes the ⟨*key*⟩=⟨*value*⟩ list for the macro ⟨*cs*⟩ to set new defaults for it (meaning the values used if you don't provide anything at use time, not those specified with the `default` secondary key). ⟨*cs*⟩ should be defined with expkvcs (so with any of the methods in subsection 1.1). Inside the ⟨*key*⟩=⟨*value*⟩ list both primary and secondary keys can be used. If ⟨*cs*⟩ was defined `\long` earlier it will still be `\long`, every other TEX prefix will be stripped (but expkvcs doesn't support them anywhere else so that should be fine). The resulting new defaults will be stored inside the ⟨*cs*⟩ locally (just as the original defaults were). If there was an unknown key forwarding added to ⟨*cs*⟩ (see subsection 1.4) any unknown key will be stored inside the list of unknown keys as well. `\ekvcChange` is not expandable!

Consider the following example:

```
\ekvcSplit\foo{a=a,b=b}{a is #1.\par b is #2.\par}
\begingroup
  \ekvcChange\foo{b=B}
  \foo{} % new defaults
  \ekvcSecondaryKeys\foo{meta c={a={#1},b={#1}}}
  \ekvcChange\foo{c=c}
  \foo{} % newer defaults
\endgroup
\foo{} % initial defaults
```

```
a is a.
b is B.
a is c.
b is c.
a is a.
b is b.
```

As a result with this the typical setup macro could be implemented:

```
\ekvcHashAndUse\foo{key=a,key=b}
\newcommand*\foosetup{\ekvcChange\foo}
```

Of course the usage is limited to a single macro `\foo`, hence this might not be as powerful as similar macros used with other ⟨*key*⟩=⟨*value*⟩ interfaces. But at least a few similar macros could be grouped using the same key parsing macro internally.

1.4 Handling Unknown Keys

If your macro should handle unknown keys without directly throwing an error you can use the special ... marker in the ⟨*primary keys*⟩ list. Since those keys will be processed

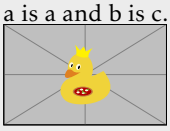
once by expkv they will be forwarded normalised: The key name will be the result of one `\detokenize`, the value will be forwarded as ␣{⟨*value*⟩}␣, so with spaces around one set of braces (this way, most other ⟨*key*⟩=⟨*value*⟩ implementations should parse the correct input).

The exact behaviour differs slightly between the two variants (as all keys do). The behaviour inside the split variants will be similar to normal primary keys, the *n*-th argument (corresponding to the position of ... inside the primary keys list) will contain any unknown key encountered while parsing the argument. And inside the split variant you can use a primary key named ... at the same time.

Example: The following will forward any unknown key to `\includegraphics` to control the appearance while processing its own keys:

```
\newcommand*\foo{\ekvoptarg\fooKV{}}
\ekvcSplitAndForward\fooKV\fooOUT
  {
    a=a
    ,...
    ,b=b
    ,...={}
  }
\newcommand\fooOUT[5]
  {%
    a is #1 and b is #3.\par
    \includegraphics[{#2}]{#5}\par
    \texttt{...} is #4.\par
  }
\foo[width=.5\linewidth, b=c,
     ...={a stupid key name, but works}]
    {example-image-duck}
```

a is a and b is c.

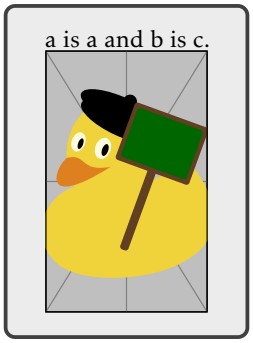


... is a stupid key name, but works.

Inside the hash variants the unknown keys list will be put inside the hash named ... (we have to use some name, so why not this). As a consequence a primary key named ... would clash with the unknown key handler. If you still used such a key it would remove any unknown key stored there until that point and replace the list with its value.

Example: The following is more or less equivalent to the example above, but with the hash variant, and it will not contain the primary ... key. We have to make sure that `\includegraphics` sees the ⟨*key*⟩=⟨*value*⟩ list, so need to expand `\ekvcValue{...}{#1}` before `\includegraphics` parses it.

```
\newcommand*\foo{\ekvoptarg\fooKV{}}
\ekvcHashAndForward\fooKV\fooOUT
  {a=a, b=b, ...}
\newcommand\fooOUT[2]
  {%
    a is \ekvcValue{a}{#1} and
    b is \ekvcValue{b}{#1}.\par
    \expanded{\noexpand\includegraphics
                          [{\ekvcValue{...}{#1}}]}
                          {#2}\par
  }
\foo[width=\linewidth, b=c]
    {example-image-duck-portrait}
```



1.5 Flags

The idea of flags is taken from expl3. They provide a way to store numerical information expandably, however only incrementing and accessing works expandably, decrementing is unexpandable. A flag has a height, which is a numerical value, and which can be raised by 1. Flags come at a high computational cost (accessing them is slow and they require more memory than normal TEX data types like registers, both getting linearly worse with the height), so don't use them if not necessary.

The state of flags is always changed locally to the current group, but not to the current macro, so if you're using one of the `t`-types involving flags bear in mind that they can affect other macros using the same flags at the current group level!

expkvics provides some macros to access, alter, and use flags. Flags of expkvics don't share a name space with the flags of expl3.

`\ekvcFlagNew` `\ekvcFlagNew⟨flag⟩`

This initialises the macro ⟨*flag*⟩ as a new flag. It isn't checked whether the macro ⟨*flag*⟩ is currently undefined. A ⟨*flag*⟩ will expand to the flag's current height with a trailing space (so you can use it directly with `\ifnum` for example and it will terminate the number scanning on its own).

All other macros dealing with flags take as a parameter a macro defined as a ⟨*flag*⟩ with `\ekvcFlagNew`.

`\ekvcFlagHeight` `\ekvcFlagHeight⟨flag⟩`

This expands to the current height of ⟨*flag*⟩ in a single step of expansion (without a trailing space).

`\ekvcFlagRaise` `\ekvcFlagRaise⟨flag⟩`

This expandably raises the height of ⟨*flag*⟩ by 1.

`\ekvcFlagSetTrue` `\ekvcFlagSetTrue⟨flag⟩`
`\ekvcFlagSetFalse`

By interpreting an even value as false and an odd value as true we can use a flag as a boolean. This expandably sets ⟨*flag*⟩ to true or false, respectively, by raising it if necessary.

`\ekvcFlagIf` `\ekvcFlagIf⟨flag⟩{⟨true⟩}{⟨false⟩}`

This interprets a ⟨*flag*⟩ as a boolean and expands to either ⟨*true*⟩ or ⟨*false*⟩.

`\ekvcFlagIfRaised` `\ekvcFlagIfRaised⟨flag⟩{⟨true⟩}{⟨false⟩}`

This tests whether the ⟨*flag*⟩ is raised, meaning it has a height greater than zero, and if so expands to ⟨*true*⟩ else to ⟨*false*⟩.

`\ekvcFlagReset` `\ekvcFlagReset⟨flag⟩`

This resets a flag (so restores its height to 0). This operation is *not* expandable and done locally. If you really intend to use flags you can reset them every now and then to keep the performance hit low.

`\ekvcFlagGetHeight` `\ekvcFlagGetHeight⟨flag⟩{⟨next⟩}`

This retrieves the current height of the ⟨*flag*⟩ and provides it as a braced argument to ⟨*next*⟩, leaving ⟨*next*⟩{⟨*height*⟩} in the input stream.

`\ekvcFlagGetHeights` `\ekvcFlagGetHeights{⟨flag-list⟩}{⟨next⟩}`

This retrieves the current height of each ⟨*flag*⟩ in the ⟨*flag-list*⟩ and provides them as a single argument to ⟨*next*⟩. Inside that argument each height is enclosed in a set of braces individually. The ⟨*flag-list*⟩ is just a single argument containing the ⟨*flag*⟩s. So a usage like `\ekvcFlagGetHeights{\myflagA\myflagB}{\stuff}` will expand to `\stuff{{⟨height-A⟩}{⟨height-B⟩}}`.

1.6 Further Examples

How could a documentation be a good documentation without enough 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` (the comments with the parameter numbers are of course not necessary and just ease reading the example).

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

The result of only the first two usages would be:

> John Doe is a Mister from the Universe. He is of any age and his hobby is to exist.
> Jonathan is a Mister from Germany. He is of a young age and his hobby is TEX coding.

Using xparse or **expkv**'s `\ekvoptarg` or `\ekvoptargTF` 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, so it would've been better to use a more feature rich ⟨*key*⟩=⟨*value*⟩ interface most likely, *e.g.*, the one provided by **expkv**|def):

```
\makeatletter
\newcommand*\nonsense{\ekvoptarg\nonsense@a{}}
\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}%
  }
\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}
```

resulting in

key	A	A	key	A	hihi	key	A	hihi	key	A	hihi
	B	B		B	B		B	A		B	A
	C	c		C	c		C	c		C	huhu
	D	d		D	d		D	d		D	d
mandatory			mandatory		haha	mandatory		hehe	mandatory		haha

1.7 Freedom for Keys!

If this was the TEXbook this subsection would have a double bend sign. Not because it is overly complicated, but because it shows things which could break expkv|cs's expandability and its alignment safety. This is for experienced users wanting to get the most flexibility and knowing what they are doing.

In case you're wondering, it is possible to define other keys than the primaries and the secondary types listed in subsection 1.2 for a macro defined with expkv|cs by using the low-level interface of expkv or even the interface provided by expkv|def. The set name used for expkv|cs's keys is the macro name, including the leading backslash, or more precisely `\string⟨cs⟩` is used. This can be exploited to define additional keys with arbitrary code. Consider the following *bad* example:

```
\ekvcSplit\foo{a=A,b=B}{a is #1.\par b is #2\par}
\protected\ekvdef{\string\foo}{c}{\def\fooC{#1}}
```

This would define a key named `c` that will store its value inside a macro. The issue with this is that this can't be done expandably. As a result, the macro `\foo` isn't always expandable any more (not that bad if this was never required; killjoy if it was) and as

soon as the key c is used, it is also no longer alignment safe[1] (might be bad depending on the usage).

So why do I show you this? Because we could as well do something useful like create a key that pre-parses the input and after that passes the parsed value on. This parsing would have to be completely expandable though. For the pass-on part we can use the following function:

\ekvcPass `\ekvcPass⟨cs⟩{⟨key⟩}{⟨value⟩}`

This passes ⟨*value*⟩ on to ⟨*key*⟩ for the expkvcs-macro ⟨*cs*⟩. It should be used inside the key parsing of a macro defined with expkvcs, else this most likely results in a low level TEX error. You can't forward anything to the special unknown key handler ... as that is no defined key.

With this we could for example split the value of a key at a hyphen and pass the parts to different keys:

```
\ekvcSplit\foo{a=A,b=B}{a is #1.\par b is #2.\par}
\ekvdef{\string\foo}{c}{\fooSplit#1\par}
\def\fooSplit#1-#2\par
  {\ekvcPass\foo{a}{#1}\ekvcPass\foo{b}{#2}}
\foo{}
\foo{c=1-2}
```

```
a is A.
b is B.
a is 1.
b is 2.
```

Additionally, there is a more general version of the `aggregate` secondary key type (described in subsection 1.2), namely the `process` key type:

process `process ⟨key⟩ = {⟨primary⟩}{⟨definition⟩}` long

This will grab the current value of a ⟨*primary*⟩ key as `#1` (without changing the current value) and the new value as `#2` and leave all the processing to ⟨*definition*⟩. You should use `\ekvcPass` to forward the values afterwards. Unlike `aggregate` you can specify whether the ⟨*key*⟩ should be long or not, this isn't inherited from the ⟨*primary*⟩ key. Keep in mind that you could easily break things here if your code does not work by expansion.

Example: We could define a key that only accepts values greater than the current value with this:

```
\ekvcSplit\foo{internal=5}{a is #1.\par}
\ekvcSecondaryKeys\foo
  {
    process a={internal}
      {%
        \ifnum#1<#2
          \ekvcPass\foo{internal}{#2}%
        \fi
      }
  }
\foo{a=1}
\foo{a=5}
\foo{a=9}
```

```
a is 5.
a is 5.
a is 9.
```

[1] This means that the ⟨*key*⟩=⟨*value*⟩-list can't contain alignment markers that are not inside an additional set of braces if used inside a TEX alignment.

1.8 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 of subsection 1.6 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 summarise this, if you have a use case for expandable ⟨*key*⟩=⟨*value*⟩ parsing macros you should go on and define them using expkv|cs. If you just want to define a simple macro with a few keys `\ekvcSplit` might be the easiest interface there is. 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.

1.9 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.

1.10 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

1.11 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 ConTeXt module

```
\writestatus{loading}{ConTeXt User Module / expkv-cs}
\usemodule[expkv]
\unprotect
\input expkv-cs.tex
\writestatus{loading}
  {ConTeXt User Module / expkv-cs / Version \ekvcVersion\space loaded}
\protect\endinput
```

2.3 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{1.2}
\def\ekvcDate{2022-02-13}
```

(*End definition for* `\ekvcVersion` *and* `\ekvcDate`. *These functions are documented on page* 17.)

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 expkvics was initialized.

`\ekvc@tripledots` This macro just serves as a marker for a comparison to allow the syntax for the unknown key handlers.

```
\def\ekvc@tripledots{...}
```

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

`\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.

```
\let\ekvc@long\ekv@empty
\let\ekvc@any@long\ekv@empty
```

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

`\ekvc@ifdefined` We want to test whether a macro is already defined. This test checks for a defined macro that isn't `\relax`.

```
\long\def\ekvc@ifdefined#1%
  {%
    \ifdefined#1%
      \ifx\relax#1%
        \ekv@fi@gobble
      \fi
      \@firstofone
      \ekv@fi@firstoftwo
    \fi
    \@secondoftwo
  }
```

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

`\ekvc@ekvset@pre@expander` `\ekvc@ekvset@pre@expander@a` `\ekvc@ekvset@pre@expander@b` This macro expands `\ekvset` twice so that the first two steps of expansion don't have to be made every time the expkvics 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. At this spot we also add `\ekv@alignsafe` and `\ekv@endalignsafe` to ensure that macros created with expkvics are alignment safe.

```
\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
  {%
    \ekv@unexpanded\expandafter{\ekv@alignsafe}%
    \ekv@unexpanded{#1}##1\ekv@unexpanded{#2}%
    \ekv@unexpanded\expandafter{\ekv@endalignsafe}%
  }
```

(*End definition for* `\ekvc@ekvset@pre@expander`, `\ekvc@ekvset@pre@expander@a`, *and* `\ekvc@ekvset@pre@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%
  {%
    \let\ekvc@helpers@needed\@firstoftwo
    \ekvc@ifdefined#1%
      {\ekvc@err@already@defined#1}%
      {\ekvcSplitAndUse@#1{}{#2}}%
  }
```

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

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

```
            \ekv@unexpanded\expandafter{\ekvc@initials}%
          }%
      }%
  }
```

(*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%
  {%
    \let\ekvc@helpers@needed\@firstoftwo
    \ekvc@ifdefined#1%
      {\ekvc@err@already@defined#1}%
      {\ekvcSplitAndUse@#1{{#2}}{#3}}%
  }
```

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

`\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%
  {%
    \let\ekvc@helpers@needed\@secondoftwo
    \ekvc@ifdefined#1%
      {\ekvc@err@already@defined#1}%
      {%
        \expandafter
        \ekvcSplitAndUse@\expandafter#1\csname ekvc@\string#1\endcsname{#2}%
        \ifnum\ekvc@keycount<1
          \ekvc@any@long\expandafter\def\csname ekvc@\string#1\endcsname{#3}%
        \else
          \ifnum\ekvc@keycount>9
            \ekvc@err@toomany{#1}%
            \let#1\ekvc@undefined
          \else
            \ekvcSplit@build@argspec
            \ekvc@any@long\expandafter
            \def\csname ekvc@\string#1\expandafter\endcsname\ekvc@tmp{#3}%
          \fi
        \fi
      }%
  }
```

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

`\ekvcSplit@build@argspec`
`\ekvcSplit@build@argspec@`

```
\protected\def\ekvcSplit@build@argspec
  {%
    \begingroup
      \edef\ekvc@tmp
        {\endgroup\def\ekv@unexpanded{\ekvc@tmp}{\ekvcSplit@build@argspec@{1}}}%
    \ekvc@tmp
  }
\def\ekvcSplit@build@argspec@#1%
```

```
  {%
    \ifnum#1>\ekvc@keycount
      \ekv@fi@gobble
    \fi
    \@firstofone
      {%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@#1\endcsname####}#1%
        \expandafter\ekvcSplit@build@argspec@\expandafter{\the\numexpr#1+1}%
      }%
  }
```

(*End definition for* `\ekvcSplit@build@argspec` *and* `\ekvcSplit@build@argspec@`.)

`\ekvc@SetupSplitKeys`
`\ekvc@SetupSplitKeys@a`
`\ekvc@SetupSplitKeys@b`
`\ekvc@SetupSplitKeys@c`
`\ekvc@SetupSplitKeys@d`
`\ekvc@SetupSplitKeys@e`
`\ekvc@SetupSplitKeys@check@unknown`
`\ekvc@SetupSplitKeys@unknown`

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=\ekv@zero
    \let\ekvc@any@long\ekv@empty
    \let\ekvc@initials\ekv@empty
    \ekvparse\ekvc@SetupSplitKeys@check@unknown\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\long\def\ekvc@SetupSplitKeys@a#1%
  {\expandafter\ekvc@SetupSplitKeys@b\detokenize{#1}\ekvc@stop}
\protected\def\ekvc@SetupSplitKeys@b#1\ekvc@stop
  {%
    \advance\ekvc@keycount1
    \let\ekvc@long\ekv@empty
    \ekvc@ifspace{#1}%
      {\ekvc@SetupSplitKeys@c#1\ekvc@stop}%
      {\ekvc@SetupSplitKeys@d{#1}}%
  }
```

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

```
\protected\long\def\ekvc@SetupSplitKeys@c#1 #2\ekvc@stop
  {%
    \ekv@ifdefined{ekvc@split@p@#1}%
      {\csname ekvc@split@p@#1\endcsname{#2}}%
      {\ekvc@SetupSplitKeys@d{#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\def\ekvc@SetupSplitKeys@d
  {%
    \begingroup\expandafter\endgroup
    \expandafter\ekvc@SetupSplitKeys@e
```

```
      \csname ekvc@splitmark@\the\ekvc@keycount\endcsname
  }
\protected\long\def\ekvc@SetupSplitKeys@e#1#2#3%
  {%
    \long\def\ekvc@tmp##1##2#1##3{##2#1{##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
      \expandafter\let\csname ekvc@\ekvc@set(#2)\endcsname\ekvc@tmp
      \edef\ekvc@tmp##1%
        {%
          \ekv@unexpanded\expandafter{\csname ekvc@\ekvc@set(#2)\endcsname}%
          {##1}%
        }%
    \fi
    \ekvlet\ekvc@set{#2}\ekvc@tmp
    \edef\ekvc@initials{\ekv@unexpanded\expandafter{\ekvc@initials#1{#3}}}%
    \ekvc@helpers@needed
      {\expandafter\ekvc@setup@splitmacro\expandafter{\the\ekvc@keycount}}%
      {}%
  }
```

If no value was provided this could either be an error, or the unknown key forwarding. We have to check this (comparing against `...` inside `\ekvc@tripledots`) and if this is the unknown key list type, set it up accordingly (advancing the key count and setting up the unknown handlers of expkv). Else we simply throw an error and ignore the incident.

```
\protected\long\def\ekvc@SetupSplitKeys@check@unknown#1%
  {%
    \begingroup
      \edef\ekvc@tmp{\detokenize{#1}}%
      \expandafter
    \endgroup
    \ifx\ekvc@tripledots\ekvc@tmp
      \advance\ekvc@keycount1
```

The `\begingroup\expandafter\endgroup` ensures that the split mark isn't actually defined (even if it just were with meaning `\relax`).

```
      \begingroup\expandafter\endgroup
      \expandafter\ekvc@SetupSplitKeys@unknown
        \csname ekvc@splitmark@\the\ekvc@keycount\endcsname
      \let\ekvc@any@long\long
    \else
      \ekvc@err@value@required{#1}%
    \fi
  }
\protected\long\def\ekvc@SetupSplitKeys@unknown#1%
  {%
    \long\expandafter\def\csname\ekv@name\ekvc@set{}u\endcsname##1##2##3#1##4%
      {##3#1{##4,##2= {##1} }}%
    \long\expandafter\def\csname\ekv@name\ekvc@set{}uN\endcsname##1##2#1##3%
      {##2#1{##3,##1}}%
```

```
    \edef\ekvc@initials{\ekv@unexpanded\expandafter{\ekvc@initials#1{}}}%
    \ekvc@helpers@needed
      {\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@d
  }
```

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

`\ekvc@split@p@short` The `short` prefix does essentially nothing, it is only provided to allow key names starting with `long` that aren't `\long`.

```
\def\ekvc@split@p@short{\ekvc@SetupSplitKeys@d}
```

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

`\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` `\ekvc@split@2` `\ekvc@split@3` `\ekvc@split@4` `\ekvc@split@5` `\ekvc@split@6` `\ekvc@split@7` Since the first few split macros are different from the others we manually set those 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\ekv@unexpanded\expandafter{\csname ekvc@split@1\endcsname}%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}%
        ##1##2##3%
      {##3{##1}##2}%
    \long\gdef\ekv@unexpanded\expandafter{\csname ekvc@split@2\endcsname}%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
        ##3##4%
      {##4{##1}{##2}##3}%
    \long\gdef\ekv@unexpanded\expandafter{\csname ekvc@split@3\endcsname}%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
        ##4##5%
      {##5{##1}{##2}{##3}##4}%
    \long\gdef\ekv@unexpanded\expandafter{\csname ekvc@split@4\endcsname}%
```

```
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@4\endcsname}##4%
        ##5##6%
      {##6{##1}{##2}{##3}{##4}##5}%
    \long\gdef\ekv@unexpanded\expandafter{\csname ekvc@split@5\endcsname}%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@4\endcsname}##4%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@5\endcsname}##5%
        ##6##7%
      {##7{##1}{##2}{##3}{##4}{##5}##6}%
    \long\gdef\ekv@unexpanded\expandafter{\csname ekvc@split@6\endcsname}%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@4\endcsname}##4%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@5\endcsname}##5%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@6\endcsname}##6%
        ##7##8%
      {##8{##1}{##2}{##3}{##4}{##5}{##6}##7}%
    \long\gdef\ekv@unexpanded\expandafter{\csname ekvc@split@7\endcsname}%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@4\endcsname}##4%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@5\endcsname}##5%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@6\endcsname}##6%
        \ekv@unexpanded\expandafter{\csname ekvc@splitmark@7\endcsname}##7%
        ##8##9%
      {##9{##1}{##2}{##3}{##4}{##5}{##6}{##7}##8}%
  }
\ekvc@tmp
\endgroup
\protected\def\ekvc@setup@splitmacro#1%
  {%
    \ekv@ifdefined{ekvc@split@#1}{}%
      {%
        \begingroup
          \edef\ekvc@tmp
            {%
              \long\gdef
                  \ekv@unexpanded\expandafter{\csname ekvc@split@#1\endcsname}%
                  ####1%
                  \ekv@unexpanded\expandafter
                    {\csname ekvc@splitmark@#1\endcsname}%
                  ####2####3%
                {%
                  \ekv@unexpanded\expandafter
                    {\csname ekvc@split@\the\numexpr#1-1\relax\endcsname}%
                  ####1{{####2}####3}%
                }%
```

```
            }%
          \ekvc@tmp
        \endgroup
      }%
  }
```

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

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

```
\protected\long\def\ekvcHashAndUse#1#2%
  {%
    \let\ekvc@helpers@needed\@firstoftwo
    \ekvc@ifdefined#1%
      {\ekvc@err@already@defined#1}%
      {\ekvcHashAndUse@#1{}{#2}}%
  }
```

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

`\ekvcHashAndUse@` 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@helpers@needed
      {%
        \ekvc@any@long\edef#1##1%
          {%
            \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}%
            \ekv@unexpanded{\ekvc@hash@pack@argument}%
            \ekv@unexpanded\expandafter{\ekvc@initials\ekvc@stop#2}%
          }%
      }%
      {%
        \ekvc@any@long\edef#1##1%
          {%
            \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}%
            \ekv@unexpanded{#2}%
            \ekv@unexpanded\expandafter{\ekvc@initials\ekvc@stop}%
          }%
      }%
  }
```

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

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

```
\protected\long\def\ekvcHashAndForward#1#2#3%
  {%
    \let\ekvc@helpers@needed\@firstoftwo
    \ekvc@ifdefined#1%
      {\ekvc@err@already@defined#1}%
      {\ekvcHashAndUse@#1{{#2}}{#3}}%
  }
```

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

`\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%
  {%
    \let\ekvc@helpers@needed\@secondoftwo
    \ekvc@ifdefined#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\ekvc@stop
          {#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` `\ekvc@SetupHashKeys@c` `\ekvc@SetupHashKeys@d` `\ekvc@SetupHashKeys@e` `\ekvc@SetupHashKeys@check@unknown` `\ekvc@SetupHashKeys@unknown` 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%
  {%
    \let\ekvc@any@long\ekv@empty
    \let\ekvc@initials\ekv@empty
    \ekvparse\ekvc@SetupHashKeys@check@unknown\ekvc@SetupHashKeys@a{#1}%
  }
\protected\long\def\ekvc@SetupHashKeys@a#1%
  {\expandafter\ekvc@SetupHashKeys@b\detokenize{#1}\ekvc@stop}
\protected\def\ekvc@SetupHashKeys@b#1\ekvc@stop
  {%
    \let\ekvc@long\ekv@empty
    \ekvc@ifspace{#1}%
      {\ekvc@SetupHashKeys@c#1\ekvc@stop}%
      {\ekvc@SetupHashKeys@d{#1}}%
  }
\protected\def\ekvc@SetupHashKeys@c#1 #2\ekvc@stop
  {%
    \ekv@ifdefined{ekvc@hash@p@#1}%
      {\csname ekvc@hash@p@#1\endcsname{#2}}%
      {\ekvc@SetupHashKeys@d{#1 #2}}%
  }
```

Again we build the marker, this time instead of a numbered one a named hashmark, inside a group to not actually define the macro used as a marker.

```
\protected\long\def\ekvc@SetupHashKeys@d#1%
  {%
```

```
    \begingroup\expandafter\endgroup
    \expandafter\ekvc@SetupHashKeys@e\csname ekvc@hashmark@#1\endcsname{#1}%
  }
```

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@e#1#2#3%
  {%
    \long\def\ekvc@tmp##1##2#1##3{##2#1{##1}}%
    \unless\ifx\ekvc@long\long
      \expandafter\let\csname ekvc@\ekvc@set(#2)\endcsname\ekvc@tmp
      \edef\ekvc@tmp##1%
        {%
          \ekv@unexpanded\expandafter{\csname ekvc@\ekvc@set(#2)\endcsname}%
          {##1}%
        }%
    \fi
    \ekvlet\ekvc@set{#2}\ekvc@tmp
    \edef\ekvc@initials{\ekv@unexpanded\expandafter{\ekvc@initials#1{#3}}}%
    \ekvc@setup@hashmacro{#2}%
  }
\protected\long\def\ekvc@SetupHashKeys@check@unknown#1%
  {%
    \begingroup
      \edef\ekvc@tmp{\detokenize{#1}}%
      \expandafter
    \endgroup
    \ifx\ekvc@tripledots\ekvc@tmp
      \ekvc@SetupHashKeys@unknown
      \let\ekvc@any@long\long
    \else
      \ekvc@err@value@required{#1}%
    \fi
  }
\def\ekvc@SetupHashKeys@unknown#1%
  {%
    \protected\def\ekvc@SetupHashKeys@unknown
      {%
        \expandafter
        \let\csname\ekv@name\ekvc@set{}u\endcsname\ekvc@hash@unknown@kv
        \expandafter
        \let\csname\ekv@name\ekvc@set{}uN\endcsname\ekvc@hash@unknown@k
        \edef\ekvc@initials{\ekv@unexpanded\expandafter{\ekvc@initials#1{}}}%
        \ekvc@setup@hashmacro{...}%
      }%
    \long\def\ekvc@hash@unknown@kv##1##2##3#1##4{##3#1{##4,##2= {##1} }}%
    \long\def\ekvc@hash@unknown@k##1##2#1##3{##2#1{##3,##1}}%
  }
\expandafter\ekvc@SetupHashKeys@unknown
  \csname ekvc@hashmark@\ekvc@tripledots\endcsname
```

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

\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@d
  }
```

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

\ekvc@hash@p@short The `short` prefix does essentially nothing, it is only provided to allow key names starting with `long` that aren't `\long`.

```
\def\ekvc@hash@p@short{\ekvc@SetupHashKeys@d}
```

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

\ekvc@setup@hashmacro The safe hash macros will be executed inside of an `\unexpanded` expansion context, so they have to insert braces 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@fasthash@#1}{}%
      {%
        \begingroup
          \edef\ekvc@tmp
            {%
              \long\gdef
                \ekv@unexpanded\expandafter{\csname ekvc@fasthash@#1\endcsname}%
                  ####1%
                  \ekv@unexpanded\expandafter
                    {\csname ekvc@hashmark@#1\endcsname}%
                  ####2####3\ekv@unexpanded{\ekvc@stop}%
                {####2}%
              \long\gdef
                \ekv@unexpanded\expandafter{\csname ekvc@safehash@#1\endcsname}%
                  ####1%
                {%
                  \ekv@unexpanded\expandafter
                    {\csname ekvc@@safehash@#1\endcsname}%
                  ####1\ekv@unexpanded{\ekvc@mark}{}%
                  \ekv@unexpanded\expandafter
                    {%
                      \csname ekvc@hashmark@#1\endcsname{}%
                      \ekvc@mark{\ekvc@err@missing@hash{#1}}\ekvc@stop
                    }%
                }%
              \long\gdef
                \ekv@unexpanded\expandafter
                  {\csname ekvc@@safehash@#1\endcsname}%
                  ####1%
                  \ekv@unexpanded\expandafter
```

```
                {\csname ekvc@hashmark@#1\endcsname}%
              ####2####3\ekv@unexpanded{\ekvc@mark}####4####5%
              \ekv@unexpanded{\ekvc@stop}%
            {%
              ####4{####2}%
            }%
          \long\gdef\ekv@unexpanded\expandafter
            {\csname ekvc@fastsplithash@#1\endcsname}%
              ####1%
              \ekv@unexpanded\expandafter
                {\csname ekvc@hashmark@#1\endcsname}%
              ####2####3\ekv@unexpanded{\ekvc@stop}####4%
            {%
              ####4{####2}%
            }%
          \long\gdef\ekv@unexpanded\expandafter
            {\csname ekvc@safesplithash@#1\endcsname}####1%
            {%
              \ekv@unexpanded\expandafter
                {\csname ekvc@@safesplithash@#1\endcsname}%
              ####1\ekv@unexpanded{\ekvc@mark\ekvc@safe@after@hash}%
              \ekv@unexpanded\expandafter
                {%
                  \csname ekvc@hashmark@#1\endcsname{}%
                  \ekvc@mark
                  {\ekvc@err@missing@hash{#1}\ekvc@safe@after@hash}%
                  \ekvc@stop
                }%
            }%
          \long\gdef\ekv@unexpanded\expandafter
            {\csname ekvc@@safesplithash@#1\endcsname}%
              ####1%
              \ekv@unexpanded\expandafter
                {\csname ekvc@hashmark@#1\endcsname}%
              ####2####3\ekv@unexpanded{\ekvc@mark}####4####5%
              \ekv@unexpanded{\ekvc@stop}%
            {%
              ####4{####2}%
            }%
        }%
      \ekvc@tmp
    \endgroup
  }%
}
```

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

`\ekvcValue` `\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%
  {%
    \ekv@unexpanded
    \expandafter\ekvcValue@\detokenize{#1}\ekvc@stop
```

```
  }
\def\ekvcValue@#1\ekvc@stop
  {%
    \ekv@ifdefined{ekvc@safehash@#1}%
      {\csname ekvc@safehash@#1\endcsname}%
      {\ekvc@err@unknown@hash{#1}\@firstoftwo{{}}}%
  }
```

(*End definition for* `\ekvcValue` *and* `\ekvcValue@`*. These functions are documented on page 5.*)

`\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@\detokenize{#1}\endcsname#2\ekvc@stop}
```

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

`\ekvcValueSplit` This splits off a single value.

`\ekvcValueSplit@`

`\ekvcValueSplit@recover`

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

(*End definition for* `\ekvcValueSplit`*,* `\ekvcValueSplit@`*, and* `\ekvcValueSplit@recover`*. These functions are documented on page 6.*)

`\ekvc@safe@after@hash`

```
\long\def\ekvc@safe@after@hash#1#2%
  {%
    #2{#1}%
  }
```

(*End definition for* `\ekvc@safe@after@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@\detokenize{#1}\endcsname#2\ekvc@stop}
```

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

`\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 `\unexpanded` expansion.

```
\long\def\ekvc@safehash@#1{\ekvc@err@empty@hash{}}
\long\def\ekvc@fasthash@#1\ekvc@stop{\ekvc@err@empty@hash}
\long\def\ekvc@safesplithash@#1#2{\ekvc@err@empty@hash#2{}}
\long\def\ekvc@fastsplithash@#1\ekvc@stop#2{\ekvc@err@empty@hash#2{}}
```

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

`\ekvcSecondaryKeys`
`\ekvcSecondaryKeys@a`
`\ekvcSecondaryKeys@b`
`\ekvcSecondaryKeys@c`

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\long\def\ekvcSecondaryKeys@a#1%
  {\expandafter\ekvcSecondaryKeys@b\detokenize{#1}\ekvc@stop}
\protected\def\ekvcSecondaryKeys@b#1\ekvc@stop
  {%
    \let\ekvc@long\ekv@empty
    \ekvc@ifspace{#1}%
      {\ekvcSecondaryKeys@c#1\ekvc@stop}%
      {\ekvc@err@missing@type{#1}\@gobble}%
  }
\protected\def\ekvcSecondaryKeys@c#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` *and others. These functions are documented on page 6.*)

`\ekvcChange`

This can be used to change the defaults of an expkvics defined macro. It checks whether there is a set with the correct name and that the macro is defined. If both is true the real work is done by `\ekvc@change`.

```
\protected\long\def\ekvcChange#1%
  {%
    \ekvifdefinedset{\string#1}%
      {%
        \ekvc@ifdefined#1%
          {\ekvc@change#1}%
          {\ekvc@err@no@key@macro#1\@gobble}%
      }%
      {\ekvc@err@no@key@macro#1\@gobble}%
  }
```

(*End definition for* `\ekvcChange`*. This function is documented on page 9.*)

`\ekvc@change`
`\ekvc@change@a`
`\ekvc@change@b`
`\ekvc@change@c`
`\ekvc@change@d`
`\ekvc@change@e`

First we need to see whether the macro is currently `\long`. For this we get the meaning and will parse it. `#1` is the macro name in which we want to change the defaults.

```
\protected\def\ekvc@change#1%
  {\expandafter\ekvc@change@a\meaning#1\ekv@stop#1}
```

A temporary definition to get the stringified `macro:`. `##1` will be the list of prefixes, we don't care for the exact contents of `##2` and `##3`.

```
\def\ekvc@change@a#1%
  {%
    \protected\def\ekvc@change@a##1#1##2->##3\ekv@stop
      {%
        \ekvc@change@iflong{##1}%
          {\ekvc@change@b{}}%
          {\ekvc@change@b{\long}}%
      }%
  }
\expandafter\ekvc@change@a\expandafter{\detokenize{macro:}}
```

Next we expand the macro once to get its contents (including the current default values with their markers) and place `\ekvc@stop` instead of an argument as a marker for the last step. `#1` is either `\long` or empty, `#2` is the macro.

```
\protected\def\ekvc@change@b#1#2%
  {\expandafter\ekvc@change@c\expandafter{#2\ekvc@stop}{#1}#2}
```

Here we place an unbalanced closing brace after the expansion of the macro. Then we just parse the ⟨*key*⟩=⟨*value*⟩-list with `\ekvset`, that will exchange the values behind the markers. Once those are changed we give control to `\ekvc@change@d`. The `\ekvset` step might horribly fail if the user defined some keys that don't behave nice. `#1` is the expansion of the macro, `#2` is either `\long` or empty, `#3` is the macro, and `#4` is the ⟨*key*⟩=⟨*value*⟩-list containing the new defaults.

```
\ekv@exparg{\protected\long\def\ekvc@change@c#1#2#3#4}%
  {%
    \expandafter\iffalse\expandafter{\expandafter{\expandafter\fi
      \ekvset{\string#3}{#4}%
      \ekvc@change@d{#2}{#3}%
      #1%
    }}%
  }
```

The final step needs to put an unbalanced opening brace after `\edef`. Also we have to protect everything from further expanding with the exception of the redefined macro's argument, which is why we placed the `\ekvc@stop` earlier. Then we need to also protect the rest of the contents from further expanding using `\unexpanded` with another unbalanced opening brace. `#1` will be either empty or `\long` and `#2` is the macro.

```
\protected\def\ekvc@change@d#1#2%
  {#1\edef#2##1{\expandafter\ekvc@change@e\iffalse}\fi}
\long\def\ekvc@change@e#1\ekvc@stop
  {\ekv@unexpanded{#1}##1\ekv@unexpanded\expandafter{\iffalse}\fi}
```

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

`\ekvc@change@iflong`
`\ekvc@change@iflong@`

Checking whether a string contains the string representation of `\long` can be done by gobbling everything up to the first `\long` and checking whether the result is completely empty. We need a temporary macro to get the result of `\string\long` inside the definitions.

```
\def\ekvc@change@iflong#1%
  {%
    \protected\def\ekvc@change@iflong##1%
      {\expandafter\ekv@ifempty\expandafter{\ekvc@change@iflong@##1#1}}%
```

```
    \def\ekvc@change@iflong@##1#1{}
  }
\expandafter\ekvc@change@iflong\expandafter{\string\long}
```

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

`\ekvcPass` This macro can be used to pass a value to a key of some macro (this way more complicated key codes are possible that in the end pass processed values on to some macro). The implemantation is pretty straight forward.

```
\long\def\ekvcPass#1#2%
  {%
    \ekvifdefined{\string#1}{#2}%
      {\csname\ekv@name{\string#1}{#2}\endcsname}%
      {\ekvc@err@unknown@key@or@macro{#1}{#2}\@gobble}%
  }
```

(*End definition for* `\ekvcPass`. *This function is documented on page* 15.)

2.3.1 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
          {%
            \ekv@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`.)

`\ekvc@t@enum` `\ekvc@type@enum` `\ekvc@h@enum` `\ekvc@h@enum@` `\ekvc@enum@name`

Enums don't need to apply special trickery to make the parts of the names retrievable, so unlike in expkv-def we don't need catcode juggling. The setup of an `enum` requires unpacking the value in two different arguments so we need an auxiliary here.

```
\def\ekvc@enum@name#1#2#3{ekvc#1(#2)#3}
\protected\long\def\ekvc@t@enum#1#2%
  {\ekvc@assert@twoargs{#2}{enum #1}{\ekvc@type@enum{#1}#2}}
\protected\long\def\ekvc@type@enum#1#2%
  {%
    \ekvifdefined\ekvc@set{#2}%
      {%
```

At run time we need another helper and we need to expand the current `\ekvc@set` now. The helper will build a control sequence from each given value, those will be set up in the `\ekvcsvloop`.

```
        \ekvc@long\edef\ekvc@tmp##1%
          {%
            \ekv@unexpanded{\expandafter\ekvc@h@enum\detokenize}{##1}%
            \ekv@unexpanded{\ekvc@stop}%
            {\ekvc@set}{#1}%
          }%
        \ekvlet\ekvc@set{#1}\ekvc@tmp
        \def\ekvc@tmp{0}%
        \expandafter\ekvcsvloop\expandafter
          {%
            \expandafter\ekvc@type@enum@
              \csname\ekv@name\ekvc@set{#2}\endcsname
              {#1}%
          }%
      }%
      {%
        \ekvc@err@unknown@key{#2}%
        \@gobble
      }%
  }
```

Here `#1` will be the key-macro of the underlying primary or secondary key, `#2` is the `enum` key's name, and `#3` will be the choice. The rest is pretty obvious.

```
\ekv@exparg{\protected\long\def\ekvc@type@enum@#1#2#3}%
  {%
    \expandafter\expandafter\expandafter\edef\expandafter
        \csname\ekvc@enum@name\ekvc@set{#2}{\detokenize{#3}}\endcsname
      {\ekv@unexpanded{#1}{\ekvc@tmp}}%
    \edef\ekvc@tmp{\the\numexpr\ekvc@tmp+1\relax}%
  }
```

The use-time helper will check if the macro for the passed in choice exists, if it doesn't throws an error, else calls that macro which will set the correct value.

```
\ekv@if@lastnamedcs
  {%
    \ekv@exparg{\def\ekvc@h@enum#1\ekvc@stop#2#3}%
      {%
        \expandafter\ifcsname\ekvc@enum@name{#2}{#3}{#1}\endcsname
          \expandafter\ekvc@h@enum@\lastnamedcs
```

```
        \fi
        \ekvc@err@unknown@enum{#2}{#3}{#1}%
      }
    \def\ekvc@h@enum@#1\fi\ekvc@err@unknown@enum#2#3#4%
      {%
        \fi
        \ifx#1\relax
          \ekvc@err@unknown@enum{#2}{#3}{#4}%
          \expandafter\@gobble
        \fi
        #1%
      }
  }
  {%
    \def\ekvc@h@enum#1%
      {%
        \def\ekvc@h@enum##1\ekvc@stop##2##3%
          {%
            \expandafter\ekvc@h@enum@
              \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname
              {##2}{##3}{##1}%
          }%
      }
    \expandafter\ekvc@h@enum\expandafter{\ekvc@enum@name{#2}{#3}{#1}}
    \def\ekvc@h@enum@#1#2#3#4%
      {%
        \ifx#1\relax
          \ekvc@err@unknown@enum{#2}{#3}{#4}%
          \expandafter\@gobble
        \fi
        #1%
      }
  }
```

We don't need `\ekvc@enum@name` anymore, so let's undefine it.

```
\let\ekvc@enum@name\ekvc@undefined
```

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

`\ekvc@t@aggregate` Aggregating isn't easy to define. We'll have to extract the correct mark for the specified key, branch correctly for short and long keys, and use a small hack to have the correct arguments on the user interface (`#1` as the current contents, `#2` as the new value). This is split into a few steps here.

First, assert that the user input is well-behaved.

```
\protected\def\ekvc@t@aggregate#1%
  {%
    \ekvc@assert@not@long{aggregate #1}%
    \ekvc@type@aggregate
      \ekvc@type@aggregate@long\ekvc@type@aggregate@short
      {process}%
      {#1}%
  }
```

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

\ekvc@type@aggregate
\ekvc@type@aggregate@a
\ekvc@type@aggregate@b

The next step stores the user defined processing in a temporary macro that's used to do the parameter number swapping later. It also builds the names of the key macro and the helper which would be used for processing a short key.

```
\protected\long\def\ekvc@type@aggregate#1#2#3#4#5%
  {%
    \ekvc@assert@twoargs{#5}{#3 #4}{\ekvc@type@aggregate@a#1#2{#4}#5}%
  }
\protected\long\def\ekvc@type@aggregate@a#1#2#3#4#5%
  {%
    \ekvifdefined\ekvc@set{#4}%
      {%
        \def\ekvc@type@aggregate@tmp##1##2{#5}%
        \begingroup\expandafter\endgroup
        \expandafter\ekvc@type@aggregate@b
          \csname\ekv@name\ekvc@set{#4}\expandafter\endcsname
          \csname ekvc@\ekvc@set(#4)\endcsname
          #1#2%
          {#3}%
      }%
      {\ekvc@err@unknown@key{#4}}%
  }
\protected\long\def\ekvc@type@aggregate@b#1#2#3#4%
  {%
    \ekvc@type@aggregate@check@long#1#2%
      {#3#1}%
      {#4#2}%
  }
```

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

\ekvc@type@aggregate@check@long
\ekvc@type@aggregate@check@long@a
\ekvc@type@aggregate@check@long@b

To check whether the primary key is long we see whether its `\meaning` contains the helper which would only be there for short keys. For this we have to get the stringified name of the internal (using `\detokenize`), and afterwards get the `\meaning` of the macro. A temporary helper does the real test by gobbling and forwarding the result to `\ekv@ifempty`.

```
\protected\long\def\ekvc@type@aggregate@check@long#1#2%
  {\expandafter\ekvc@type@aggregate@check@long@a\detokenize{#2}\ekv@stop#1}
\protected\long\def\ekvc@type@aggregate@check@long@a#1\ekv@stop#2%
  {%
    \def\ekvc@type@aggregate@check@long@@##1#1{}%
    \expandafter\ekvc@type@aggregate@check@long@b\meaning#2\ekv@stop{#1}%
  }
\protected\def\ekvc@type@aggregate@check@long@b#1\ekv@stop#2%
  {\expandafter\ekv@ifempty\expandafter{\ekvc@type@aggregate@check@long@@#1#2}}
```

(*End definition for* `\ekvc@type@aggregate@check@long`, `\ekvc@type@aggregate@check@long@a`, *and* `\ekvc@type@aggregate@che`

\ekvc@type@aggregate@long
\ekvc@type@aggregate@long@

The long variant just builds the split mark we extract, uses the hack to swap argument order, and then does the definition via `\ekvlet` and a temporary macro.

```
\protected\long\def\ekvc@type@aggregate@long#1%
  {%
    \begingroup\expandafter\endgroup\expandafter
    \ekvc@type@aggregate@long@
      \csname\ekvc@extract@mark#1\expandafter\endcsname
```

```
      \expandafter{\ekvc@type@aggregate@tmp{##3}{##1}}%
  }
\protected\long\def\ekvc@type@aggregate@long@#1#2#3%
  {%
    \long\def\ekvc@type@aggregate@tmp##1##2#1##3{##2#1{#2}}
    \ekvlet\ekvc@set{#3}\ekvc@type@aggregate@tmp
  }
```

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

`\ekvc@type@aggregate@short`
`\ekvc@type@aggregate@short@`

The short variant will have to build the marker and the name of the helper function, and swap the user argument order. Hence here are a few more `\expandafter`s involved. But afterwards we can do the definition of the key and the helper macro directly.

```
\protected\long\def\ekvc@type@aggregate@short#1#2%
  {%
    \begingroup\expandafter\endgroup\expandafter
    \ekvc@type@aggregate@short@
      \csname\ekvc@extract@mark#1\expandafter\endcsname
      \csname ekvc@\ekvc@set(#2)\expandafter\endcsname
      \expandafter{\ekvc@type@aggregate@tmp{##3}{##1}}%
      {#2}%
  }
\protected\long\def\ekvc@type@aggregate@short@#1#2#3#4%
  {%
    \ekvdef\ekvc@set{#4}{#2{##1}}%
    \long\def#2##1##2#1##3{##2#1{#3}}%
  }
```

(*End definition for* `\ekvc@type@aggregate@short` *and* `\ekvc@type@aggregate@short@`.)

`\ekvc@t@process`

The `process` type can reuse much of `aggregate`, just the last step of definition differ.

```
\protected\def\ekvc@t@process
  {%
    \ifx\ekvc@long\long
      \ekv@fi@firstoftwo
    \fi
    \@secondoftwo
      {%
        \ekvc@type@aggregate
          \ekvc@type@process@long\ekvc@type@process@long
      }%
      {%
        \ekvc@type@aggregate
          \ekvc@type@process@short\ekvc@type@process@short
      }%
      {process}%
  }
```

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

`\ekvc@type@process@long`
`\ekvc@type@process@long@`

This defines a temporary macro to grab the current value (found after the marker `#1`), executes the user code and puts everything back to where it belongs. Then `\ekvlet` is used to assign that meaning to the key macro.

```
\protected\long\def\ekvc@type@process@long#1%
```

```
  {%
    \begingroup\expandafter\endgroup\expandafter
    \ekvc@type@process@long@
      \csname\ekvc@extract@mark#1\expandafter\endcsname
      \expandafter{\ekvc@type@aggregate@tmp{##3}{##1}}%
  }
\protected\long\def\ekvc@type@process@long@#1#2#3%
  {%
    \long\def\ekvc@type@aggregate@tmp##1##2#1##3{#2##2#1{##3}}%
    \ekvlet\ekvc@set{#3}\ekvc@type@aggregate@tmp
  }
```

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

`\ekvc@type@process@short`
`\ekvc@type@process@short@`

We define the key macro directly to just grab the argument once and forward it to the auxiliary. That one does essentially the same as the long variant.

```
\protected\long\def\ekvc@type@process@short#1#2%
  {%
    \begingroup\expandafter\endgroup\expandafter
    \ekvc@type@process@short@
      \csname\ekvc@extract@mark#1\expandafter\endcsname
      \csname ekvc@\ekvc@set(#2)\expandafter\endcsname
      \expandafter{\ekvc@type@aggregate@tmp{##3}{##1}}%
      {#2}%
  }
\protected\long\def\ekvc@type@process@short@#1#2#3#4%
  {%
    \ekvdef\ekvc@set{#4}{#2{##1}}%
    \long\def#2##1##2#1##3{#3##2#1{##3}}%
  }
```

(*End definition for* `\ekvc@type@process@short` *and* `\ekvc@type@process@short@`.)

`\ekvc@t@flag-bool`

```
\protected\expandafter\def\csname ekvc@t@flag-bool\endcsname#1#2%
  {%
    \ekvc@assert@not@long{flag-bool #1}%
    \unless\ifdefined#2\ekvcFlagNew#2\fi
    \ekvdef\ekvc@set{#1}%
      {%
        \ekv@ifdefined{ekvc@flag@set@##1}%
          {%
            \csname ekvc@flag@set@##1\expandafter\endcsname
              \ekvcFlagHeight#2\ekv@stop#2%
          }%
          {\ekvc@err@invalid@bool{##1}}%
      }%
  }
```

(*End definition for* `\ekvc@t@flag-bool`.)

`\ekvc@t@flag-true`
`\ekvc@t@flag-false`
`\ekvc@t@flag-raise`
`\ekvc@type@flag`

```
\protected\def\ekvc@type@flag#1#2#3#4%
  {%
    \ekvc@assert@not@long{flag-#1 #3}%
```

```
    \unless\ifdefined#4\ekvcFlagNew#4\fi
    \ekv@exparg{\ekvdefNoVal\ekvc@set{#3}}{#2#4}%
  }
\protected\expandafter\def\csname ekvc@t@flag-true\endcsname
  {\ekvc@type@flag{true}\ekvcFlagSetTrue}
\protected\expandafter\def\csname ekvc@t@flag-false\endcsname
  {\ekvc@type@flag{false}\ekvcFlagSetFalse}
\protected\expandafter\def\csname ekvc@t@flag-raise\endcsname
  {\ekvc@type@flag{raise}\ekvcFlagRaise}
```

(*End definition for* `\ekvc@t@flag-true` *and others.*)

2.3.2 Flags

The basic idea of flags is to store information by the fact that TEX expandably assigns the meaning `\relax` to undefined control sequences which were built with `\csname`. This mechanism is borrowed from expl3.

`\ekvc@flag@name` `\ekvc@flag@namescheme`

Flags follow a simple naming scheme which we define here. `\ekvc@flag@name` will store the name of an internal function that is used to build names of the second naming scheme defined by `\ekvc@flag@namescheme`.

```
\def\ekvc@flag@name{ekvcf\string}
\def\ekvc@flag@namescheme#1#2{ekvch#2#1}
```

(*End definition for* `\ekvc@flag@name` *and* `\ekvc@flag@namescheme`.)

`\ekvcFlagHeight`

For semantic reasons we use `\number` with another name.

```
\let\ekvcFlagHeight\number
```

(*End definition for* `\ekvcFlagHeight`. *This function is documented on page* 11.)

`\ekvcFlagNew`

This macro defines a new flag. It stores the function build with the `\ekvc@flag@name` naming scheme after the internal function `\ekvc@flag@height` that'll determine the current flag height. It'll also define the macro named via `\ekvc@flag@name` to build names according to `\ekvc@flag@namescheme`.

```
\protected\def\ekvcFlagNew#1%
  {%
    \edef#1%
      {%
        \ekv@unexpanded{\ekvc@flag@height}%
          \ekv@unexpanded\expandafter{\csname\ekvc@flag@name#1\endcsname}%
      }%
    \ekv@expargtwice
      {\expandafter\def\csname\ekvc@flag@name#1\endcsname##1}%
      {\expandafter\ekvc@flag@namescheme\expandafter{\string#1}{##1}}%
  }
```

(*End definition for* `\ekvcFlagNew`. *This function is documented on page* 11.)

`\ekvc@flag@height` `\ekvc@flag@height@`

This macro gets the height of a flag by a simple loop. The first loop iteration differs a bit from the following in that it doesn't have to get the current iteration count. The space at the end of `\ekvc@flag@height` ends the `\number` evaluation.

```
\def\ekvc@flag@height#1%
  {%
```

```
    \ifcsname#10\endcsname
      \ekvc@flag@height@1\ekv@stop#1%
    \fi
    \@firstofone{0} % leave this space
  }
\def\ekvc@flag@height@#1\ekv@stop#2\fi\@firstofone#3%
  {%
    \fi
    \ifcsname#2{#1}\endcsname
      \expandafter\ekvc@flag@height@\the\numexpr#1+1\relax\ekv@stop#2%
    \fi
    \@firstofone{#1}%
  }
```

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

`\ekvcFlagRaise` Raising a flag simply means letting the `\ekvc@flag@namescheme` macro for the current height to relax. The result of raising a flag is that its height is bigger by 1.

```
\ekv@exparg{\def\ekvcFlagRaise#1}%
  {%
    \expandafter\expandafter\expandafter\@gobble\expandafter
      \csname\ekvc@flag@namescheme{\string#1}{\ekvcFlagHeight#1}\endcsname
  }
```

(*End definition for* `\ekvcFlagRaise`. *This function is documented on page* 11.)

`\ekvcFlagSetTrue` `\ekvcFlagSetFalse` `\ekvc@flag@set@true` `\ekvc@flag@set@false` A flag is considered true if its current height is odd, and as false if it is even. Therefore `\ekvcFlagSetTrue` and `\ekvcFlagSetFalse` only need to raise the flag if the opposing boolean value is the current one.

```
\def\ekvcFlagSetTrue#1%
  {\expandafter\ekvc@flag@set@true\ekvcFlagHeight#1\ekv@stop#1}
\def\ekvcFlagSetFalse#1%
  {\expandafter\ekvc@flag@set@false\ekvcFlagHeight#1\ekv@stop#1}
```

We can expand `\ekvc@flag@namescheme` at definition time here, which is why we're using a temporary definition to set up `\ekvc@flag@set@true` and `\ekvc@flag@set@false`.

```
\def\ekvc@flag@set@true#1%
  {%
    \def\ekvc@flag@set@true##1\ekv@stop##2%
      {%
        \ifodd##1
          \ekv@fi@gobble
        \fi
        \@firstofone{\expandafter\@gobble\csname#1\endcsname}%
      }%
    \def\ekvc@flag@set@false##1\ekv@stop##2%
      {%
        \ifodd##1
          \ekv@fi@firstofone
        \fi
        \@gobble{\expandafter\@gobble\csname#1\endcsname}%
      }%
  }
\expandafter\ekvc@flag@set@true\expandafter
  {\ekvc@flag@namescheme{\string#2}{#1}}
```

(*End definition for* `\ekvcFlagSetTrue` *and others. These functions are documented on page 11.*)

`\ekvcFlagIf` As already explained, truthiness is defined as a flag's height being odd, so we just branch accordingly here.

```
\def\ekvcFlagIf#1%
  {%
    \ifodd#1%
      \ekv@fi@firstoftwo
    \fi
    \@secondoftwo
  }
```

(*End definition for* `\ekvcFlagIf`*. This function is documented on page 12.*)

`\ekvcFlagIfRaised` This macro uses flags as a switch, if a flag's current height is bigger than 0 this test yields true.

```
\ekv@exparg{\def\ekvcFlagIfRaised#1}%
  {%
    \expandafter\ifcsname\ekvc@flag@namescheme{\string#1}0\endcsname
      \ekv@fi@firstoftwo
    \fi
    \@secondoftwo
  }
```

(*End definition for* `\ekvcFlagIfRaised`*. This function is documented on page 12.*)

`\ekvcFlagReset` `\ekvc@flag@reset` `\ekvc@flag@reset@` Resetting works by locally letting all the defined internal macros named after `\ekvc@flag@namescheme` to undefined.

```
\protected\def\ekvcFlagReset#1%
  {\expandafter\ekvc@flag@reset\csname\ekvc@flag@name#1\endcsname}
\protected\def\ekvc@flag@reset#1%
  {%
    \ifcsname#10\endcsname
      \expandafter\let\csname#10\endcsname\ekvc@undefined
      \ekvc@flag@reset@1\ekv@stop#1%
    \fi
  }
\protected\def\ekvc@flag@reset@#1\ekv@stop#2\fi
  {%
    \fi
    \ifcsname#2{#1}\endcsname
      \expandafter\let\csname#2{#1}\endcsname\ekvc@undefined
      \expandafter\ekvc@flag@reset@\the\numexpr#1+1\relax\ekv@stop#2%
    \fi
  }
```

(*End definition for* `\ekvcFlagReset`*,* `\ekvc@flag@reset`*, and* `\ekvc@flag@reset@`*. These functions are documented on page 12.*)

`\ekvcFlagGetHeight` `\ekvc@flag@get@height@single` These are just small helpers, first getting the height of the flag and then passing it on to the user supplied code.

```
\def\ekvcFlagGetHeight#1%
  {\expandafter\ekvc@flag@get@height@single\ekvcFlagHeight#1\ekv@stop}
\long\def\ekvc@flag@get@height@single#1\ekv@stop#2{#2{#1}}
```

(*End definition for* `\ekvcFlagGetHeight` *and* `\ekvc@flag@get@height@single`*. These functions are documented on page 12.*)

`\ekvcFlagGetHeights`
`\ekvc@flag@get@heights`
`\ekvc@flag@get@heights@`
`\ekvc@flag@get@heights@done`

This works by a simple loop that stops at `\ekv@stop`. As long as that marker isn't hit, get the next flags height and put it into a list after `\ekv@stop`. `\ekvc@flag@get@heights@` uses the same marker name for the end of the height, which shouldn't clash in any case. Once we're done we remove the remainder of the current iteration and leave the user supplied code in the input stream with all the flags' heights as a single argument.

```
\def\ekvcFlagGetHeights#1%
  {%
    \ekvc@flag@get@heights#1\ekv@stop{}%
  }
\def\ekvc@flag@get@heights#1%
  {%
    \ekv@gobbleto@stop#1\ekvc@flag@get@heights@done\ekv@stop
    \expandafter\ekvc@flag@get@heights@\ekvcFlagHeight#1\ekv@stop
  }
\def\ekvc@flag@get@heights@#1\ekv@stop#2\ekv@stop#3%
  {\ekvc@flag@get@heights#2\ekv@stop{#3{#1}}}
\long\def\ekvc@flag@get@heights@done
    \ekv@stop
    \expandafter\ekvc@flag@get@heights@\ekvcFlagHeight\ekv@stop\ekv@stop#1#2%
  {#2{#1}}
```

(*End definition for* `\ekvcFlagGetHeights` *and others. These functions are documented on page 12.*)

2.3.3 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@`*.*)

`\ekvc@ifnottwoargs`
`\ekvc@ifempty@gtwo`

Used to test whether a token list contains exactly two TEX arguments.

```
\long\def\ekvc@ifnottwoargs#1%
  {%
    \ekvc@ifempty@gtwo#1\ekv@ifempty@B
      \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
  }
\long\def\ekvc@ifempty@gtwo#1#2{\ekv@ifempty@\ekv@ifempty@A}
```

(*End definition for* `\ekvc@ifnottwoargs` *and* `\ekvc@ifempty@gtwo`*.*)

`\ekvc@extract@mark`
`\ekvc@extract@mark@`

This is used to extract the mark of a split or hash key from its definition. This is kind of fragile, it assumes `#1` is always a macro used for hashing or splitting. Also it assumes that the escape character is a backslash.

```
\def\ekvc@extract@mark#1{\expandafter\ekvc@extract@mark@\meaning#1\ekv@stop}
\begingroup
\lccode`;=`#
\lccode`/=`\\
\lowercase{\endgroup
\def\ekvc@extract@mark@#1:#2/#3 ;#4\ekv@stop{#3}%
}
```

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

2.3.4 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`.)

`\ekvc@assert@twoargs`

Some keys need exactly two arguments as their definition, so we have to somehow assert this.

```
\protected\long\def\ekvc@assert@twoargs#1#2%
  {\ekvc@ifnottwoargs{#1}{\ekvc@err@not@two{#2}}}
```

(*End definition for* `\ekvc@assert@twoargs`.)

2.3.5 Messages

`\ekvc@err@toomany`
`\ekvc@err@value@required`
`\ekvc@err@missing@type`
`\ekvc@err@already@defined`
`\ekvc@err@no@key@macro`
`\ekvc@err@not@two`

Boring unexpandable error messages.

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

```
\protected\long\def\ekvc@err@unknown@key#1%
  {%
    \errmessage
      {expkv-cs Error: Unknown key `\ekv@unexpanded{#1}' for macro `\ekvc@set'}%
  }
\protected\long\def\ekvc@err@no@key@macro#1%
  {\errmessage{expkv-cs Error: \string#1 is no key=val macro}}
\protected\long\def\ekvc@err@not@two#1%
  {%
    \errmessage
      {%
        expkv-cs Error: Definition of `\ekv@unexpanded{#1}' doesn't contain
        exactly two arguments%
      }%
  }
```

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

`\ekvc@err` We need a way to throw error messages expandably in some contexts.

```
\ekv@exparg{\long\def\ekvc@err#1}{\ekverr{expkv-cs}{#1}}
```

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

`\ekvc@err@unknown@hash` `\ekvc@err@empty@hash` `\ekvc@err@missing@hash` `\ekvc@err@invalid@bool` `\ekvc@err@unknown@enum` 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}}
\def\ekvc@err@invalid@bool#1{\ekvc@err{invalid boolean value `#1'}}
\long\def\ekvc@err@unknown@key@or@macro#1#2%
  {\ekvc@err{unknown key `#2' for macro #1}}
\def\ekvc@err@unknown@enum#1#2#3%
  {\ekvc@err{unknown value `#3' for enum `#2' in macro #1}}
```

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

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.

U

W