

exp_kv|CS

define expandable $\langle key \rangle = \langle value \rangle$ macros using exp_kv

Jonathan P. Spratte*

2021-05-24 v0.9

Abstract

exp_kv|CS provides two small interfaces to define expandable $\langle key \rangle = \langle value \rangle$ macros using exp_kv. It therefore lowers the entrance boundary to expandable $\langle key \rangle = \langle value \rangle$ macros. The stylised name is exp_kv|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

1	Documentation	2
1.1	Define Macros and Primary Keys	2
1.1.1	Primary Keys	2
1.1.2	Split	3
1.1.3	Hash	3
1.2	Secondary Keys	5
1.2.1	p-type Prefixes	5
1.2.2	t-type Prefixes	5
1.3	Flags	6
1.4	Example	7
1.5	Speed Considerations	10
1.6	Useless Macros	11
1.7	Bugs	11
1.8	License	11
2	Implementation	12
2.1	The L ^A T _E X Package	12
2.2	The Generic Code	12
2.2.1	Secondary Key Types	25
2.2.2	Flags	27
2.2.3	Helper Macros	31
2.2.4	Assertions	31
2.2.5	Messages	31
	Index	33

*jspratte@yahoo.de

1 Documentation

The `expkv` package enables the new possibility of creating $\langle key \rangle = \langle value \rangle$ macros which are fully expandable. The creation of such macros is however cumbersome for the average user. `expkvics` tries to step in here. It provides interfaces to define $\langle key \rangle = \langle value \rangle$ macros without worrying too much about the implementation. In case you're wondering now, the `cs` in `expkvics` stands for control sequence, because `def` was already taken by `expkvDEF` 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 `expkvics` 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 $\langle value \rangle$ by using a hash. Well, actually there is no real hash, just some markers which are parsed, but this shouldn't be apparent to the user, the behaviour matches that of a hash-table.

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

A word of advice you should consider: If your macro doesn't have to be expandable (and often it doesn't) don't use `expkvics`. The interface has some overhead (though it still can be considered fast – check [subsection 1.5](#)) 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 `expkvDEF` for convenience. Or you resort to another $\langle key \rangle = \langle value \rangle$ interface.

`expkvics` is usable as generic code and as a *LaTeX* package. It'll automatically load `expkv` in the same mode as well. To use it, just use one of

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

1.1 Define Macros and Primary Keys

All macros defined with `expkvics` have to be previously undefined or have the `\meaning` of `\relax`. This is necessary as there is no way to undefine keys once they are set up (neither `expkv` nor `expkvics` 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 $\langle primary\ keys \rangle$. This argument should be a $\langle key \rangle = \langle value \rangle$ list where each $\langle key \rangle$ will be one primary key and $\langle value \rangle$ the associated initial value. By default all keys are defined short, but you can define long keys by prefixing $\langle key \rangle$ with `long` (e.g., `long name=Jonathan P. Spratte`). You only need `long` if the key should be able to take a `\par` token. Note however that long keys are a microscopic grain faster (due to some internals of `expkvics`). Only if at least one of the keys was long the $\langle cs \rangle$ in the following defining macros will be `\long`. For obvious reasons there is no possibility to define a macro or key as `\protected`.

At the moment `\expkvcs` 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.

<code>\ekvcSplit</code>	<code>\ekvcSplit<cs>{<primary keys>}{<definition>}</code>
-------------------------	---

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.

<code>\ekvcSplitAndForward</code>	<code>\ekvcSplitAndForward<cs>{<after>}{<primary keys>}</code>
-----------------------------------	--

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}
```

<code>\ekvcSplitAndUse</code>	<code>\ekvcSplitAndUse<cs>{<primary keys>}</code>
-------------------------------	---

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

```
<cs>{<key>=<value>, ...}{<after>}
```

1.1.3 Hash

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

<hr/> <hr/>	<code>\ekvcHash</code>	<code>\ekvcHash<cs>{\<primary keys>}{\<definition>}</code>	This defines <code><cs></code> to be a macro taking one mandatory argument which should contain a <code><key>=<value></code> 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 <code><definition></code> , in which you can access the <code><value></code> of a <code><key></code> by using <code>\ekvcValue{\<key>}{#1}</code> .
<hr/> <hr/>	<code>\ekvcHashAndForward</code>	<code>\ekvcHashAndForward<cs>{\<after>}{\<primary keys>}</code>	This defines <code><cs></code> to be a macro taking one mandatory argument which should contain a <code><key>=<value></code> 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 <code><after></code> . You can use a single macro for <code><after></code> or use some arbitrary stuff, which will be left in the input stream before the hashed <code><key>=<value></code> list with one set of braces stripped. In the macro called in <code><after></code> you can access the <code><value></code> of a <code><key></code> by using <code>\ekvcValue{\<key>}{#1}</code> (or whichever argument the hashed <code><key>=<value></code> list will be).
<hr/> <hr/>	<code>\ekvcHashAndUse</code>	<code>\ekvcHashAndUse<cs>{\<primary keys>}</code>	This will roughly do the same as <code>\ekvcHashAndForward</code> , but instead of specifying what will be used after hashing the keys, <code><cs></code> will use what follows the <code><key>=<value></code> list. So its syntax will be $\langle cs \rangle \{ \langle key \rangle = \langle value \rangle, \dots \} \{ \langle after \rangle \}$
<hr/> <hr/>	<code>\ekvcValue</code>	<code>\ekvcValue{\<key>}{\<key list>}</code>	This is a safe (but slow) way to access your keys in a hash variant. <code><key></code> is the key which's <code><value></code> you want to use out of the <code><key list></code> . <code><key list></code> should be the key list argument forwarded to your underlying macro by <code>\ekvcHash</code> , <code>\ekvcHashAndForward</code> , or <code>\ekvcHashAndUse</code> . It will be tested whether the hash function to access that <code><key></code> exists, the <code><key></code> argument is not empty, and that the <code><key list></code> really contains a <code><value></code> of that <code><key></code> . This macro needs exactly two steps of expansion.
<hr/> <hr/>	<code>\ekvcValueFast</code>	<code>\ekvcValueFast{\<key>}{\<key list>}</code>	This behaves just like <code>\ekvcValue</code> , but <i>without any</i> safety tests. As a result this is about 1.4 times faster <i>but</i> will throw low level T _E X errors eventually if the hash function isn't defined or the <code><key></code> isn't part of the <code><key list></code> (e.g., because it was defined as a key for another macro – all macros share the same hash function per <code><key></code>). Use it if you know what you're doing. This macro needs exactly three steps of expansion in the no-errors case.
<hr/> <hr/>	<code>\ekvcValueSplit</code>	<code>\ekvcValueSplit{\<key>}{\<key list>}{\<next>}</code>	If you need a specific <code><key></code> from a <code><key list></code> 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 <code>\ekvcValueSplit</code> will extract one specific <code><key></code> 's value from the list and forward the remainder of the list as the first and the <code><key></code> 's value as the second argument to <code><next></code> , so the result of this will be <code><next>{\<key list'\>}{\<value>}</code> with <code><key list'\></code> the remaining list. This is almost as fast as <code>\ekvcValue</code> and runs the same tests. Keep in mind that you can't fetch for the same <code><key></code> again from <code><key list'\></code> as it got removed.

\ekvcValueSplitFast

`\ekvcValueSplitFast{<key>}{<key list>}{<next>}`

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

1.2 Secondary Keys

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

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

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

\ekvcSecondaryKeys

`\ekvcSecondaryKeys<cs>{<key>=<value>, ...}`

This is the front facing macro to define secondary keys. For the macro `<cs>` define `<key>` to have definition `<value>`. The general syntax for `<key>` should be

`<prefix> <name>`

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

1.2.1 p-type Prefixes

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

long

The following key will be defined `\long`.

1.2.2 t-type Prefixes

If you're familiar with `expkvDEF` 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.

<hr/> <hr/> alias	alias $\langle key \rangle = \langle key_2 \rangle$	long
	This assigns the definition of $\langle key_2 \rangle$ to $\langle key \rangle$. As a result $\langle key \rangle$ is an alias for $\langle key_2 \rangle$ behaving just the same. Both the value taking and the NoVal version (that's expl_v slang for a key not accepting a value) will be copied if they are defined when alias is used. Of course, $\langle key_2 \rangle$ has to be defined, be it as a primary or secondary one.	
<hr/> <hr/> default	default $\langle key \rangle = \{\langle default \rangle\}$	long
	If $\langle key \rangle$ is a defined value taking key, you can define a NoVal version with this that will behave as if $\langle key \rangle$ was given $\langle default \rangle$ as its $\langle value \rangle$. Note that this doesn't change the initial values of primary keys set at definition time in <code>\ekvcSplit</code> and friends. $\langle key \rangle$ can be a primary or secondary key.	
<hr/> <hr/> flag-bool	flag-bool $\langle key \rangle = \langle cs \rangle$	long
	This is a secondary key that doesn't directly involve any of the primary or secondary keys. This defines $\langle key \rangle$ to take a value, which should be either true or false, and set the flag called $\langle cs \rangle$ accordingly as a boolean. If $\langle cs \rangle$ isn't defined yet it will be initialised as a flag. Please also read subsection 1.3 .	
<hr/> <hr/> flag-true flag-false	flag-true $\langle key \rangle = \langle cs \rangle$	long
	This is a secondary key that doesn't directly involve any of the primary or secondary keys. This defines $\langle key \rangle$ to take no value and set the flag called $\langle cs \rangle$ to true or false, respectively. If $\langle cs \rangle$ isn't defined yet it will be initialised as a flag. Please also read subsection 1.3 .	
<hr/> <hr/> flag-raise	flag-raise $\langle key \rangle = \langle cs \rangle$	long
	This is a secondary key that doesn't directly involve any of the primary or secondary keys. This defines $\langle key \rangle$ to take no value and raise the flag called $\langle cs \rangle$. If $\langle cs \rangle$ isn't defined yet it will be initialised as a flag. Please also read subsection 1.3 .	

1.3 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 **T_EX** 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 at the current group level!

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

<hr/> <hr/> \ekvcFlagNew	\ekvcFlagNew(<i>flag</i>)
	This initialises the macro $\langle flag \rangle$ as a new flag. It isn't checked whether the macro $\langle flag \rangle$ is currently undefined. A $\langle flag \rangle$ will expand to the flag's current height with a trailing space (so you can use it directly with <code>\ifnum</code> for example and it will terminate the number scanning itself).

All other macros dealing with flags take as a parameter a macro defined as a $\langle flag \rangle$ with `\ekvcFlagNew`.

<u><code>\ekvcFlagHeight</code></u>	<code>\ekvcFlagHeight$\langle flag \rangle$</code> This expands to the current height of $\langle flag \rangle$ in a single step of expansion (without a trailing space).
<u><code>\ekvcFlagRaise</code></u>	<code>\ekvcFlagRaise$\langle flag \rangle$</code> This expandably raises the height of $\langle flag \rangle$ by 1.
<u><code>\ekvcFlagSetTrue</code></u> <u><code>\ekvcFlagSetFalse</code></u>	<code>\ekvcFlagSetTrue$\langle flag \rangle$</code> By interpreting an even value as false and an odd value as true we can use a flag as a boolean. This expandably sets $\langle flag \rangle$ to true or false, respectively, by raising it if necessary.
<u><code>\ekvcFlagIf</code></u>	<code>\ekvcFlagIf$\langle flag \rangle${$\langle true \rangle$}{$\langle false \rangle$}</code> This interprets a $\langle flag \rangle$ as a boolean and expands to either $\langle true \rangle$ or $\langle false \rangle$.
<u><code>\ekvcFlagIfRaised</code></u>	<code>\ekvcFlagIfRaised$\langle flag \rangle${$\langle true \rangle$}{$\langle false \rangle$}</code> This tests whether the $\langle flag \rangle$ is raised, meaning it has a height greater than zero, and if so expands to $\langle true \rangle$ else to $\langle false \rangle$.
<u><code>\ekvcFlagReset</code></u>	<code>\ekvcFlagReset$\langle flag \rangle$</code> This resets a flag (so restores its height to 0). This operation is <i>not</i> 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.
<u><code>\ekvcFlagGetHeight</code></u>	<code>\ekvcFlagGetHeight$\langle flag \rangle${$\langle next \rangle$}</code> This retrieves the current height of the $\langle flag \rangle$ and provides it as a braced argument to $\langle next \rangle$, leaving $\langle next \rangle$ { $\langle height \rangle$ } in the input stream.
<u><code>\ekvcFlagGetHeights</code></u>	<code>\ekvcFlagGetHeights{$\langle flag-list \rangle$}{$\langle next \rangle$}</code> This retrieves the current height of each $\langle flag \rangle$ in the $\langle flag-list \rangle$ and provides them as a single argument to $\langle next \rangle$. Inside that argument each height is enclosed in a set of braces individually. The $\langle flag-list \rangle$ is just a single argument containing the $\langle flag \rangle$ s. So a usage like <code>\ekvcFlagGetHeights{\myflagA\myflagB}{\stuff}</code> will expand to <code>\stuff{{$\langle height-A \rangle$}}{$\langle height-B \rangle$}}</code> .

1.4 Example

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

```

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

```

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

```

\ekvcSecondaryKeys\character
{
  alias pro = pronoun,
  alias pos = possessive,
  nmeta me =
  {
    name=Jonathan,
    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 $\langle key \rangle = \langle value \rangle$ list.

The result of all 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 T_EX coding.

Using `xparse` or `expkv`'s `\ekvoptarg` or `\ekvoptargTF` and forwarding arguments one can easily define $\langle key \rangle = \langle value \rangle$ 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):

```
\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

1.5 Speed Considerations

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

The speed considerations in this subsection use the first example in this documentation as the benchmark. So we have seven keys and a short sentence which should be typeset. For comparisons I use the following equivalent `expkvDEF` definitions. Each result is the average between changing no keys from their initial values and altering four. Furthermore I'll compare three variants of `expkvics` with the `expkvDEF` 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 `expkvDEF` 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 `expkvDEF` 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 $\langle key \rangle = \langle value \rangle$ parsing part and only 1.5 to 2.8 times faster if the typesetting part is factored in. So the real choke isn't the parsing.

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

1.6 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.

<code>\ekvcDate</code>
<code>\ekvcVersion</code>

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

1.7 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

1.8 License

Copyright © 2020–2021 Jonathan P. Spratte

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

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

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

2 Implementation

2.1 The L^AT_EX Package

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

```
1 \RequirePackage{expkv}
2 \def\ekvc@tmp
3   {%
4     \ProvidesFile{expkv-cs.tex}%
5     [%
6       \ekvcDate\space v\ekvcVersion\space
7       define expandable key=val macros using expkv%
8     ]%
9   }
10 \input{expkv-cs.tex}
11 \ProvidesPackage{expkv-cs}%
12 [%
13   \ekvcDate\space v\ekvcVersion\space
14   define expandable key=val macros using expkv%
15 ]
```

2.2 The Generic Code

The rest of this implementation will be the generic code.

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

```
16 \input expkv
    We make sure that expkv-cs.tex is only input once:
17 \expandafter\ifx\csname ekvcVersion\endcsname\relax
18 \else
19   \expandafter\endinput
20 \fi
```

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

```
\ekvcDate
21 \def\ekvcVersion{0.9}
22 \def\ekvcDate{2021-05-24}
```

(End definition for `\ekvcVersion` and `\ekvcDate`. These functions are documented on page 11.)

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

```
23 \csname ekvc@tmp\endcsname
    Store the category code of @ to later be able to reset it and change it to 11 for now.
24 \expandafter\chardef\csname ekvc@tmp\endcsname=\catcode'\@
25 \catcode'\@=11
```

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

`\ekvc@zero` There are different ways to define a `\romannumeral` expansion. If the control is never given to user input, the fastest way is to just execute your code and use a `\dimendef` zero to end it afterwards. If the control is given to user input (so user input should be expanded) the fastest and best way is to use `\romannumeral'\^~@`, this will expand the user input until something unexpandable is found or a space (which would then be gobbled). However, we want to use the former approach since we don't want to expand any user input, just make sure own code is expanded in two steps. Since we want to use the `\dimendef` variant, we have to actually provide such a token. Since both plain \TeX and \LaTeX define `\z@` we can use that (but we use a private name for it). Luckily, `expkvics` already contains `\ekv@zero`, which we can use in `expkvics`.

(End definition for \ekvc@zero.)

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

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

(End definition for \ekvc@keycount.)

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

`\ekvc@any@long`

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

```
28 \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`.

```
29 \long\def\ekvc@ifdefined#1%
```

```
30 {%
```

```
31   \ifdefined#1%
```

```
32     \ifx\relax#1%
```

```
33       \ekv@fi@gobble
```

```
34     \fi
```

```
35     \@firstofone
```

```
36     \ekv@fi@firstoftwo
```

```
37   \fi
```

```
38   \@secondoftwo
```

```
39 }
```

(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.

```
40 \def\ekvc@ekvset@pre@expander#1%
```

```
41 {%
```

```
42   \expandafter\ekvc@ekvset@pre@expander@a\ekvset{#1}\ekvc@stop\ekvc@stop
```

```
43 }
```

```
44 \def\ekvc@ekvset@pre@expander@a
```

```

45     {%
46     \expandafter\ekvc@ekvset@pre@expander@b
47     }
48 \def\ekvc@ekvset@pre@expander@b#1\ekvc@stop#2\ekvc@stop
49     {%
50     \unexpanded\expandafter{\ekv@alignsafe}%
51     \unexpanded{#1}##1\unexpanded{#2}%
52     \unexpanded\expandafter{\ekv@endalignsafe}%
53     }

```

(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.

```

54 \protected\long\def\ekvcSplitAndUse#1#2%
55     {%
56     \let\ekvc@helpers@needed\@firstoftwo
57     \ekvc@ifdefined#1%
58     {\ekvc@err@already@defined#1}%
59     {\ekvcSplitAndUse@#1}{#2}}%
60     }

```

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

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

```

61 \protected\long\def\ekvcSplitAndUse@#1#2#3%
62     {%
63     \edef\ekvc@set{\string#1}%
64     \ekvc@SetupSplitKeys{#3}%
65     \ekvc@helpers@needed
66     {%
67     \ekvc@any@long\edef#1##1%
68     {%
69     \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}%
70     \unexpanded\expandafter
71     {\csname ekvc@split@the\ekvc@keycount\endcsname}%
72     \unexpanded\expandafter{\ekvc@initials}{#2}%
73     }%
74     }%
75     {%
76     \ekvc@any@long\edef#1##1%
77     {%
78     \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}%
79     \unexpanded{#2}%
80     \unexpanded\expandafter{\ekvc@initials}%
81     }%
82     }%
83     }

```

(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.

```

84 \protected\long\def\ekvcSplitAndForward#1#2#3%
85 {%
86   \let\ekvc@helpers@needed\@firstoftwo
87   \ekvc@ifdefined#1%
88     {\ekvc@err@already@defined#1}%
89     {\ekvcSplitAndUse@#1#{#2}#{#3}}%
90 }

```

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

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

```

91 \protected\long\def\ekvcSplit#1#2#3%
92 {%
93   \let\ekvc@helpers@needed\@secondoftwo
94   \ekvc@ifdefined#1%
95     {\ekvc@err@already@defined#1}%
96     {%
97       \expandafter
98       \ekvcSplitAndUse@\expandafter#1\csname ekvc@\string#1\endcsname{#2}%
99       \ifnum\ekvc@keycount<1
100         \ekvc@any@long\expandafter\def\csname ekvc@\string#1\endcsname{#3}%
101       \else
102         \ifnum\ekvc@keycount>9
103           \ekvc@err@toomany{#1}%
104           \let#1\ekvc@undefined
105         \else
106           \ekvcSplit@build@argspec
107           \ekvc@any@long\expandafter
108           \def\csname ekvc@\string#1\expandafter\endcsname\ekvc@tmp{#3}%
109         \fi
110       \fi
111     }%
112 }

```

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

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

```

113 \protected\def\ekvcSplit@build@argspec
114 {%
115   \begingroup
116   \edef\ekvc@tmp
117     {\endgroup\def\unexpanded{\ekvc@tmp}{\ekvcSplit@build@argspec@{1}}}%
118   \ekvc@tmp
119 }
120 \def\ekvcSplit@build@argspec@#1%
121 {%
122   \ifnum#1>\ekvc@keycount
123     \ekv@fi@gobble
124   \fi
125   \@firstofone
126   {%

```

```

127         \unexpanded\expandafter{\csname ekvc@splitmark@#1\endcsname####}#1%
128         \expandafter\ekvcSplit@build@argspec@\expandafter{\the\numexpr#1+1}%
129     }%
130 }

```

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

\ekvc@SetupSplitKeys
\ekvc@SetupSplitKeys@a
\ekvc@SetupSplitKeys@b
\ekvc@SetupSplitKeys@c

These macros parse the list of keys and set up the key macros. First we need to initialise some macros and start \ekvparsed.

```

131 \protected\long\def\ekvc@SetupSplitKeys
132 {%
133     \ekvc@keycount=0
134     \let\ekvc@any@long\ekv@empty
135     \let\ekvc@initials\ekv@empty
136     \ekvparsed\ekvc@err@value@required\ekvc@SetupSplitKeys@a
137 }

```

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.

```

138 \protected\def\ekvc@SetupSplitKeys@a#1%
139 {%
140     \advance\ekvc@keycount1
141     \let\ekvc@long\ekv@empty
142     \ekvc@ifspace{#1}%
143     {\ekvc@SetupSplitKeys@b#1\ekvc@stop}%
144     {\ekvc@SetupSplitKeys@c{#1}}%
145 }

```

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

```

146 \protected\def\ekvc@SetupSplitKeys@b#1 #2\ekvc@stop
147 {%
148     \ekv@ifdefined{ekvc@split@p@#1}%
149     {\csname ekvc@split@p@#1\endcsname{#2}}%
150     {\ekvc@SetupSplitKeys@c{#1 #2}}%
151 }

```

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.

```

152 \protected\long\def\ekvc@SetupSplitKeys@c#1#2%
153 {%
154     \begingroup
155     \edef\ekvc@tmp
156     {%
157         \endgroup
158         \long\def\unexpanded{\ekvc@tmp}####1####2%
159         \unexpanded\expandafter
160         {\csname ekvc@splitmark@\the\ekvc@keycount\endcsname}####3%
161     }%
162     ####2%
163     \unexpanded\expandafter

```



```

164         {\csname ekvc@splitmark@the\ekvc@keycount\endcsname}{####1}%
165     }%

```

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.

```

166     \unless\ifx\ekvc@long\long
167     \let\unexpanded\expandafter
168     {\csname ekvc@\ekvc@set{#1}\endcsname\ekvc@tmp}%
169     \def\unexpanded{\ekvc@tmp}####1%
170     {%
171         \unexpanded\expandafter{\csname ekvc@\ekvc@set{#1}\endcsname}%
172         {####1}%
173     }%
174     \fi
175     \def\unexpanded{\ekvc@initials}%
176     {%
177         \unexpanded\expandafter{\ekvc@initials}%
178         \unexpanded\expandafter
179         {\csname ekvc@splitmark@the\ekvc@keycount\endcsname{#2}}}%
180     }%
181 }%
182 \ekvc@tmp
183 \ekvlet\ekvc@set{#1}\ekvc@tmp
184 \ekvc@helpers@needed
185 {\expandafter\ekvc@setup@splitmacro\expandafter{\the\ekvc@keycount}}}%
186 {}%
187 }

```

(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.

```

188 \protected\def\ekvc@split@p@long
189     {%
190         \let\ekvc@long\long
191         \let\ekvc@any@long\long
192         \ekvc@SetupSplitKeys@c
193     }

```

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

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

```

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

```

(End definition for \ekvc@defarggobbler.)

\ekvc@setup@splitmacro 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.

```

195 \begingroup
\ekvc@split@1
\ekvc@split@2
\ekvc@split@3
\ekvc@split@4
\ekvc@split@5
\ekvc@split@6
\ekvc@split@7

```

```

196 \edef\ekvc@tmp
197 {%
198   \long\gdef\unexpanded\expandafter{\csname ekvc@split@1\endcsname}%
199   \unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}%
200   ##1##2##3%
201   {##3{##1}##2}%
202   \long\gdef\unexpanded\expandafter{\csname ekvc@split@2\endcsname}%
203   \unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
204   \unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
205   ##3##4%
206   {##4{##1}{##2}##3}%
207   \long\gdef\unexpanded\expandafter{\csname ekvc@split@3\endcsname}%
208   \unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
209   \unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
210   \unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
211   ##4##5%
212   {##5{##1}{##2}{##3}##4}%
213   \long\gdef\unexpanded\expandafter{\csname ekvc@split@4\endcsname}%
214   \unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
215   \unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
216   \unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
217   \unexpanded\expandafter{\csname ekvc@splitmark@4\endcsname}##4%
218   ##5##6%
219   {##6{##1}{##2}{##3}{##4}##5}%
220   \long\gdef\unexpanded\expandafter{\csname ekvc@split@5\endcsname}%
221   \unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
222   \unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
223   \unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
224   \unexpanded\expandafter{\csname ekvc@splitmark@4\endcsname}##4%
225   \unexpanded\expandafter{\csname ekvc@splitmark@5\endcsname}##5%
226   ##6##7%
227   {##7{##1}{##2}{##3}{##4}{##5}##6}%
228   \long\gdef\unexpanded\expandafter{\csname ekvc@split@6\endcsname}%
229   \unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
230   \unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
231   \unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
232   \unexpanded\expandafter{\csname ekvc@splitmark@4\endcsname}##4%
233   \unexpanded\expandafter{\csname ekvc@splitmark@5\endcsname}##5%
234   \unexpanded\expandafter{\csname ekvc@splitmark@6\endcsname}##6%
235   ##7##8%
236   {##8{##1}{##2}{##3}{##4}{##5}{##6}##7}%
237   \long\gdef\unexpanded\expandafter{\csname ekvc@split@7\endcsname}%
238   \unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1%
239   \unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2%
240   \unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3%
241   \unexpanded\expandafter{\csname ekvc@splitmark@4\endcsname}##4%
242   \unexpanded\expandafter{\csname ekvc@splitmark@5\endcsname}##5%
243   \unexpanded\expandafter{\csname ekvc@splitmark@6\endcsname}##6%
244   \unexpanded\expandafter{\csname ekvc@splitmark@7\endcsname}##7%
245   ##8##9%
246   {##9{##1}{##2}{##3}{##4}{##5}{##6}{##7}##8}%
247 }
248 \ekvc@tmp
249 \endgroup

```

```

250 \protected\def\ekvc@setup@splitmacro#1%
251 {%
252   \ekvc@ifdefined{ekvc@split@#1}{}%
253   {%
254     \begingroup
255     \edef\ekvc@tmp
256     {%
257       \long\gdef
258       \unexpanded\expandafter{\csname ekvc@split@#1\endcsname}%
259       ###1%
260       \unexpanded\expandafter{\csname ekvc@splitmark@#1\endcsname}%
261       ###2###3%
262     }%
263     \unexpanded\expandafter
264     {\csname ekvc@split@the\numexpr#1-1\relax\endcsname}%
265     ###1{{###2}###3}%
266   }%
267 }%
268 \ekvc@tmp
269 \endgroup
270 }%
271 }

```

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

\ekvcHashAndUse \ekvcHashAndUse works just like \ekvcSplitAndUse.

```

272 \protected\long\def\ekvcHashAndUse#1#2%
273 {%
274   \let\ekvc@helpers@needed\@firstoftwo
275   \ekvc@ifdefined#1%
276   {\ekvc@err@already@defined#1}%
277   {\ekvcHashAndUse@#1}{#2}}%
278 }

```

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

\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.

```

279 \protected\long\def\ekvcHashAndUse@#1#2#3%
280 {%
281   \edef\ekvc@set{\string#1}%
282   \ekvc@SetupHashKeys{#3}%
283   \ekvc@helpers@needed
284   {%
285     \ekvc@any@long\edef#1##1%
286     {%
287       \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}%
288       \unexpanded{\ekvc@hash@pack@argument}%
289       \unexpanded\expandafter{\ekvc@initials\ekvc@stop#2}%
290     }%
291   }%
292   {%
293     \ekvc@any@long\edef#1##1%

```

```

294         {%
295         \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}%
296         \unexpanded{#2}%
297         \unexpanded\expandafter{\ekvc@initials\ekvc@stop}%
298         }%
299     }%
300 }

```

(End definition for \ekvcHashAndUse@.)

\ekvcHashAndForward \ekvcHashAndForward works just like \ekvcSplitAndForward.

```

301 \protected\long\def\ekvcHashAndForward#1#2#3%
302     {%
303     \let\ekvc@helpers@needed\@firstoftwo
304     \ekvc@ifdefined#1%
305     {\ekvc@err@already@defined#1}%
306     {\ekvcHashAndUse@#1{#2}{#3}}%
307     }

```

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

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

```

308 \protected\long\def\ekvcHash#1#2#3%
309     {%
310     \let\ekvc@helpers@needed\@secondoftwo
311     \ekvc@ifdefined#1%
312     {\ekvc@err@already@defined#1}%
313     {%
314     \expandafter
315     \ekvcHashAndUse@\expandafter#1\csname ekvc@\string#1\endcsname{#2}%
316     \ekvc@any@long\expandafter\def\csname ekvc@\string#1\endcsname
317     ##1\ekvc@stop
318     {#3}%
319     }%
320     }

```

(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.

```

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

```

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

\ekvc@SetupHashKeys 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.

\ekvc@SetupHashKeys@a
\ekvc@SetupHashKeys@b

```

322 \protected\long\def\ekvc@SetupHashKeys#1%
323     {%
324     \let\ekvc@any@long\ekv@empty
325     \let\ekvc@initials\ekv@empty
326     \ekvparse\ekvc@err@value@required\ekvc@SetupHashKeys@a{#1}%
327     }
328 \protected\def\ekvc@SetupHashKeys@a#1%
329     {%

```

```

330 \let\ekvc@long\ekv@empty
331 \ekvc@ifspace{#1}%
332   {\ekvc@SetupHashKeys@b#1\ekvc@stop}%
333   {\ekvc@SetupHashKeys@c{#1}}%
334 }
335 \protected\def\ekvc@SetupHashKeys@b#1 #2\ekvc@stop
336 {%
337   \ekvc@ifdefined{ekvc@hash@p@#1}%
338   {\csname ekvc@hash@p@#1\endcsname{#2}}%
339   {\ekvc@SetupHashKeys@c{#1 #2}}%
340 }

```

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`.

```

341 \protected\long\def\ekvc@SetupHashKeys@c#1#2%
342 {%
343   \begingroup
344   \edef\ekvc@tmp
345     {%
346       \endgroup
347       \long\def\unexpanded{\ekvc@tmp}####1####2%
348       \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}####3%
349       {%
350         ####2%
351         \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}{####1}%
352       }%
353       \unless\ifx\ekvc@long\long
354         \let\unexpanded\expandafter
355         {\csname ekvc@\ekvc@set{#1}\endcsname\ekvc@tmp}%
356         \def\unexpanded{\ekvc@tmp}####1%
357         {%
358           \unexpanded\expandafter{\csname ekvc@\ekvc@set{#1}\endcsname}%
359           {####1}%
360         }%
361       \fi
362       \def\unexpanded{\ekvc@initials}%
363       {%
364         \unexpanded\expandafter{\ekvc@initials}%
365         \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname{#2}}%
366       }%
367     }%
368   \ekvc@tmp
369   \ekvlet\ekvc@set{#1}\ekvc@tmp
370   \ekvc@setup@hashmacro{#1}%
371 }

```

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

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

```

372 \protected\def\ekvc@hash@p@long
373 {%
374   \let\ekvc@long\long

```

```

375 \let\ekvc@any@long\long
376 \ekvc@SetupHashKeys@c
377 }

```

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

\ekvc@setup@hashmacro

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

```

378 \protected\def\ekvc@setup@hashmacro#1%
379 {%
380   \ekv@ifdefined{ekvc@hash@#1}{}%
381   {%
382     \begingroup
383     \edef\ekvc@tmp
384     {%
385       \long\gdef
386       \unexpanded\expandafter{\csname ekvc@fasthash@#1\endcsname}%
387       ####1%
388       \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}%
389       ####2####3\unexpanded{\ekvc@stop}%
390       {####2}%
391     \long\gdef
392     \unexpanded\expandafter{\csname ekvc@safefasthash@#1\endcsname}%
393     ####1%
394     {%
395       \unexpanded\expandafter{\csname ekvc@@safefasthash@#1\endcsname}%
396       ####1\unexpanded{\ekvc@mark\ekv@zero}%
397       \unexpanded\expandafter
398       {%
399         \csname ekvc@hashmark@#1\endcsname\ekv@zero
400         \ekvc@mark{\ekvc@err@missing@hash{#1}}\ekvc@stop
401       }%
402     }%
403     \long\gdef
404     \unexpanded\expandafter{\csname ekvc@@safefasthash@#1\endcsname}%
405     ####1%
406     \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}%
407     ####2####3\unexpanded{\ekvc@mark}####4####5%
408     \unexpanded{\ekvc@stop}%
409     {%
410       ####4####2%
411     }%
412     \long\gdef\unexpanded\expandafter
413     {\csname ekvc@fastsplithash@#1\endcsname}%
414     ####1%
415     \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}%
416     ####2####3\unexpanded{\ekvc@stop}####4%
417     {%
418       ####4{####1####3}{####2}%
419     }%

```

```

420 \long\gdef\unexpanded\expandafter
421 {\csname ekvc@safesplithash@#1\endcsname}###1%
422 {%
423 \unexpanded\expandafter
424 {\csname ekvc@@safesplithash@#1\endcsname}%
425 ###1\unexpanded{\ekvc@mark\ekvc@safe@found@hash}%
426 \unexpanded\expandafter
427 {%
428 \csname ekvc@hashmark@#1\endcsname}%
429 \ekvc@mark{\ekvc@err@missing@hash{#1}\ekvc@safe@no@hash}%
430 \ekvc@stop
431 }%
432 }%
433 \long\gdef\unexpanded\expandafter
434 {\csname ekvc@@safesplithash@#1\endcsname}%
435 ###1%
436 \unexpanded\expandafter{\csname ekvc@hashmark@#1\endcsname}%
437 ###2###3\unexpanded{\ekvc@mark}###4###5%
438 \unexpanded{\ekvc@stop}%
439 {%
440 ###4{###2}###1###3\unexpanded{\ekvc@stop}%
441 }%
442 }%
443 \ekvc@tmp
444 \endgroup
445 }%
446 }

```

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

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

```

447 \long\def\ekvcValue#1%
448 {%
449 \romannumeral
450 \ekv@ifdefined{\ekvc@safehash@#1}%
451 {\csname ekvc@safehash@#1\endcsname}%
452 {\ekvc@err@unknown@hash{#1}\@firstoftwo\ekv@zero}%
453 }

```

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

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

```

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

```

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

\ekvcValueSplit This splits off a single value.

```

\ekvcValueSplit@recover
455 \long\def\ekvcValueSplit#1%
456 {%
457 \ekv@ifdefined{\ekvc@safesplithash@#1}%
458 {\csname ekvc@safesplithash@#1\endcsname}%
459 {\ekvc@err@unknown@hash{#1}\ekvcValueSplit@recover}%
460 }

```

```

461 \long\def\ekvcValueSplit@recover#1#2%
462     {%
463         #2{#1}{}%
464     }

```

(End definition for \ekvcValueSplit and \ekvcValueSplit@recover. These functions are documented on page 4.)

```

\ekvc@safe@found@hash
\ekvc@safe@no@hash
465 \long\def\ekvc@safe@found@hash#1#2\ekvc@stop#3%
466     {%
467         #3{#2}{#1}%
468     }
469 \long\def\ekvc@safe@no@hash#1#2\ekvc@mark\ekvc@safe@found@hash\ekvc@stop#3%
470     {%
471         #3{#2}{}%
472     }

```

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

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

```

473 \long\def\ekvcValueSplitFast#1#2%
474     {%
475         \csname ekvc@fastsplithash@#1\endcsname#2\ekvc@stop
476     }

```

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

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

```

477 \long\def\ekvc@safehash@#1{\ekvc@err@empty@hash\ekv@zero}
478 \long\def\ekvc@fasthash@#1\ekvc@stop{\ekvc@err@empty@hash}
479 \long\def\ekvc@safesplithash@#1#2{\ekvc@err@empty@hash#2{#1}{}}
480 \long\def\ekvc@fastsplithash@#1\ekvc@stop#2{\ekvc@err@empty@hash#2{#1}{}}

```

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

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

```

481 \protected\long\def\ekvcSecondaryKeys#1#2%
482     {%
483         \edef\ekvc@set{\string#1}%
484         \ekvparse\ekvc@err@value@required\ekvcSecondaryKeys@a{#2}%
485     }
486 \protected\def\ekvcSecondaryKeys@a#1%
487     {%
488         \let\ekvc@long\ekv@empty
489         \ekvc@ifspace{#1}%
490         {\ekvcSecondaryKeys@b#1\ekvc@stop}%
491         {\ekvc@err@missing@type{#1}\@gobble}%

```



```

492 }
493 \protected\def\ekvcSecondaryKeys@b#1 #2\ekvc@stop
494 {%
495   \ekv@ifdefined{ekvc@p@#1}%
496   {\csname ekvc@p@#1\endcsname}%
497   {%
498     \ekv@ifdefined{ekvc@t@#1}%
499     {\csname ekvc@t@#1\endcsname}%
500     {\ekvc@err@unknown@keytype{#1}\@firstoftwo\@gobble}%
501   }%
502   {#2}%
503 }

```

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

2.2.1 Secondary Key Types

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

```

504 \protected\def\ekvc@p@long#1%
505 {%
506   \ekvc@ifspace{#1}%
507   {%
508     \let\ekvc@long\long
509     \ekvc@after@ptype#1\ekvc@stop
510   }%
511   {\ekvc@err@missing@type{long #1}\@gobble}%
512 }
513 \protected\def\ekvc@after@ptype#1 #2\ekvc@stop
514 {%
515   \ekv@ifdefined{ekvc@t@#1}%
516   {\csname ekvc@t@#1\endcsname{#2}}%
517   {\ekvc@err@unknown@keytype{#1}\@gobble}%
518 }

```

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

`\ekvc@t@meta` The meta and nmeta key types use a nested `\ekvset` to set other keys in the same macro's `<set>`.
`\ekvc@t@nmeta`

```

519 \protected\def\ekvc@t@meta
520 {%
521   \edef\ekvc@tmp{\ekvc@set}%
522   \expandafter\ekvc@type@meta\expandafter{\ekvc@tmp}\ekvc@long{##1}\ekvlet
523 }
524 \protected\def\ekvc@t@nmeta#1%
525 {%
526   \ekvc@assert@not@long{nmeta #1}%
527   \edef\ekvc@tmp{\ekvc@set}%
528   \expandafter\ekvc@type@meta\expandafter{\ekvc@tmp}{-}\ekvletNoVal{#1}%
529 }
530 \protected\long\def\ekvc@type@meta#1#2#3#4#5#6%
531 {%
532   \expandafter\ekvc@type@meta@a\expandafter{\ekvset{#1}{#6}}{#2}{#3}%
533   #4\ekvc@set{#5}\ekvc@tmp

```

```

534 }
535 \protected\def\ekvc@type@meta@a
536 {%
537   \expandafter\ekvc@type@meta@b\expandafter
538 }
539 \protected\long\def\ekvc@type@meta@b#1#2#3%
540 {%
541   #2\def\ekvc@tmp#3{#1}%
542 }

```

(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.

```

543 \protected\def\ekvc@t@alias#1#2%
544 {%
545   \ekvc@assert@not@long{alias #1}%
546   \let\ekvc@tmp\@firstofone
547   \ekvifdefined\ekvc@set{#2}%
548   {%
549     \ekvletkv\ekvc@set{#1}\ekvc@set{#2}%
550     \let\ekvc@tmp\@gobble
551   }%
552   {}%
553   \ekvifdefinedNoVal\ekvc@set{#2}%
554   {%
555     \ekvletkvNoVal\ekvc@set{#1}\ekvc@set{#2}%
556     \let\ekvc@tmp\@gobble
557   }%
558   {}%
559   \ekvc@tmp{\ekvc@err@unknown@key{#2}}%
560 }

```

(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.

```

561 \protected\long\def\ekvc@t@default#1#2%
562 {%
563   \ekvifdefined\ekvc@set{#1}%
564   {%
565     \ekvc@assert@not@long{default #1}%
566     \edef\ekvc@tmp
567     {%
568       \unexpanded\expandafter
569       {\csname\ekv@name\ekvc@set{#1}\endcsname{#2}}%
570     }%
571     \ekvletNoVal\ekvc@set{#1}\ekvc@tmp
572   }%
573   {\ekvc@err@unknown@key{#1}}%
574 }

```

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

`\ekvc@t@flag-bool`

```

575 \protected\expandafter\def\csname ekvc@t@flag-bool\endcsname#1#2%
576 {%
577   \ekvc@assert@not@long{flag-bool #1}%
578   \unless\ifdefined#2\ekvcFlagNew#2\fi
579   \ekvdef\ekvc@set{#1}%
580   {%
581     \ekv@ifdefined{ekvc@flag@set@##1}%
582     {%
583       \csname ekvc@flag@set@##1\expandafter\endcsname
584       \ekvcFlagHeight#2\ekv@stop#2%
585     }%
586     {\ekvc@err@invalid@bool{##1}}%
587   }%
588 }

```

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

`\ekvc@t@flag-true`

`\ekvc@t@flag-false`

`\ekvc@t@flag-raise`

`\ekvc@type@flag`

```

589 \protected\def\ekvc@type@flag#1#2#3#4%
590 {%
591   \ekvc@assert@not@long{flag-#1 #3}%
592   \unless\ifdefined#4\ekvcFlagNew#4\fi
593   \ekv@exparg{\ekvdefNoVal\ekvc@set{#3}}{#2#4}%
594 }
595 \protected\expandafter\def\csname ekvc@t@flag-true\endcsname
596 {\ekvc@type@flag{true}\ekvcFlagSetTrue}
597 \protected\expandafter\def\csname ekvc@t@flag-false\endcsname
598 {\ekvc@type@flag{false}\ekvcFlagSetFalse}
599 \protected\expandafter\def\csname ekvc@t@flag-raise\endcsname
600 {\ekvc@type@flag{raise}\ekvcFlagRaise}

```

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

2.2.2 Flags

The basic idea of flags is to store information by the fact that T_EX 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`.

```

601 \def\ekvc@flag@name{ekvcf\string}
602 \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.

```

603 \let\ekvcFlagHeight\number

```

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

\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.

```

604 \protected\def\ekvcFlagNew#1%
605 {%
606   \edef#1%
607   {%
608     \unexpanded{\ekvc@flag@height}%
609     \unexpanded\expandafter{\csname\ekvc@flag@name#1\endcsname}%
610   }%
611   \ekv@expargtwice
612   {\expandafter\def\csname\ekvc@flag@name#1\endcsname##1}%
613   {\expandafter\ekvc@flag@namescheme\expandafter{\string#1}{##1}}%
614 }

```

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

\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.

```

615 \def\ekvc@flag@height#1%
616 {%
617   \ifcsname#1\endcsname
618   \ekvc@flag@height@1\ekv@stop#1%
619   \fi
620   \@firstofone{0} % leave this space
621 }
622 \def\ekvc@flag@height@#1\ekv@stop#2\fi\@firstofone#3%
623 {%
624   \fi
625   \ifcsname#2{#1}\endcsname
626   \expandafter\ekvc@flag@height@\the\numexpr#1+1\relax\ekv@stop#2%
627   \fi
628   \@firstofone{#1}%
629 }

```

(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.

```

630 \ekv@exparg{\def\ekvcFlagRaise#1}%
631 {%
632   \expandafter\expandafter\expandafter\@gobble\expandafter
633   \csname\ekvc@flag@namescheme{\string#1}{\ekvcFlagHeight#1}\endcsname
634 }

```

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

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

```

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

```

```

637 \def\ekvcFlagSetFalse#1%
638   {\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`.

```

639 \def\ekvc@flag@set@true#1%
640   {%
641     \def\ekvc@flag@set@true##1\ekv@stop##2%
642       {%
643         \ifodd##1
644           \ekv@fi@gobble
645         \fi
646         \@firstofone{\expandafter\@gobble\csname#1\endcsname}%
647       }%
648     \def\ekvc@flag@set@false##1\ekv@stop##2%
649       {%
650         \ifodd##1
651           \ekv@fi@firstofone
652         \fi
653         \@gobble{\expandafter\@gobble\csname#1\endcsname}%
654       }%
655   }
656 \expandafter\ekvc@flag@set@true\expandafter
657   {\ekvc@flag@namescheme{\string#2}{#1}}

```

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

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

```

658 \def\ekvcFlagIf#1%
659   {%
660     \ifodd#1%
661       \ekv@fi@firstoftwo
662     \fi
663     \@secondoftwo
664   }

```

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

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

```

665 \ekv@exparg{\def\ekvcFlagIfRaised#1}%
666   {%
667     \expandafter\ifcsname\ekvc@flag@namescheme{\string#1}0\endcsname
668     \ekv@fi@firstoftwo
669   \fi
670   \@secondoftwo
671   }

```

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

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

```

\ekvc@flag@reset
\ekvc@flag@reset@
672 \protected\def\ekvcFlagReset#1%
673   {\expandafter\ekvc@flag@reset\csname\ekvc@flag@name#1\endcsname}

```

```

674 \protected\def\ekvc@flag@reset#1%
675 {%
676   \ifcsname#10\endcsname
677   \expandafter\let\csname#10\endcsname\ekvc@undefined
678   \ekvc@flag@reset@1\ekv@stop#1%
679   \fi
680 }
681 \protected\def\ekvc@flag@reset@#1\ekv@stop#2\fi
682 {%
683   \fi
684   \ifcsname#2{#1}\endcsname
685   \expandafter\let\csname#2{#1}\endcsname\ekvc@undefined
686   \expandafter\ekvc@flag@reset@\the\numexpr#1+1\relax\ekv@stop#2%
687   \fi
688 }

```

(End definition for `\ekvcFlagReset`, `\ekvc@flag@reset`, and `\ekvc@flag@reset@`. These functions are documented on page 7.)

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

`\ekvc@flag@get@height@single`

```

689 \def\ekvcFlagGetHeight#1%
690 {%\expandafter\ekvc@flag@get@height@single\ekvcFlagHeight#1\ekv@stop}
691 \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 7.)

`\ekvcFlagGetHeights` 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.

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

```

692 \def\ekvcFlagGetHeights#1%
693 {%
694   \ekvc@flag@get@heights#1\ekv@stop{}%
695 }
696 \def\ekvc@flag@get@heights#1%
697 {%
698   \ekv@gobbleto@stop#1\ekvc@flag@get@heights@done\ekv@stop
699   \expandafter\ekvc@flag@get@heights@\ekvcFlagHeight#1\ekv@stop
700 }
701 \def\ekvc@flag@get@heights@#1\ekv@stop#2\ekv@stop#3%
702 {%\ekvc@flag@get@heights#2\ekv@stop{#3{#1}}}%
703 \long\def\ekvc@flag@get@heights@done
704   \ekv@stop
705   \expandafter\ekvc@flag@get@heights@\ekvcFlagHeight\ekv@stop\ekv@stop#1#2%
706   {#2{#1}}

```

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

2.2.3 Helper Macros

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

```
\ekvc@ifspace@
\ekvc@ifspace@
707 \long\def\ekvc@ifspace#1%
708   {%
709     \ekvc@ifspace@#1 \ekv@ifempty@B
710     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
711   }
712 \long\def\ekvc@ifspace@#1 % keep this space
713   {%
714     \ekv@ifempty@\ekv@ifempty@A
715   }
```

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

2.2.4 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.

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

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

2.2.5 Messages

Boring unexpandable error messages.

```
\ekvc@err@toomany
\ekvc@err@value@required
\ekvc@err@missing@type
\ekvc@err@already@defined
717 \protected\def\ekvc@err@toomany#1%
718   {%
719     \errmessage{expkv-cs Error: Too many keys for macro '\string#1'}%
720   }
721 \protected\def\ekvc@err@value@required#1%
722   {%
723     \errmessage{expkv-cs Error: Missing value for key '\unexpanded{#1}'}%
724   }
725 \protected\def\ekvc@err@missing@type#1%
726   {%
727     \errmessage
728       {expkv-cs Error: Missing type for secondary key '\unexpanded{#1}'}%
729   }
730 \protected\def\ekvc@err@no@long#1%
731   {%
732     \errmessage
733       {expkv-cs Error: prefix 'long' not accepted for '\unexpanded{#1}'}%
734   }
735 \protected\def\ekvc@err@already@defined#1%
736   {%
737     \errmessage{expkv-cs Error: Macro '\string#1' already defined}%
738   }
739 \protected\def\ekvc@err@unknown@keytype#1%
740   {%
741     \errmessage{expkv-cs Error: Unknown key type '\unexpanded{#1}'}%
742   }
743 \protected\def\ekvc@err@unknown@key#1%
744   {%
```

```

745     \errmessage
746     {expkv-cs Error: Unknown key ‘\unexpanded{#1}’ for macro ‘\ekvc@set’}%
747   }

```

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

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

```

748 \ekv@exparg{\long\def\ekvc@err#1}{\ekvrr{expkv-cs}{#1}}

```

(End definition for \ekvc@err.)

`\ekvc@err@unknown@hash` And here are the expandable error messages.

```

\ekvc@err@empty@hash 749 \long\def\ekvc@err@unknown@hash#1{\ekvc@err{unknown hash ‘#1’}}
\ekvc@err@missing@hash 750 \long\def\ekvc@err@missing@hash#1{\ekvc@err{hash ‘#1’ not found}}
\ekvc@err@invalid@bool 751 \long\def\ekvc@err@empty@hash{\ekvc@err{empty hash}}
752 \def\ekvc@err@invalid@bool#1{\ekvc@err{invalid boolean value ‘#1’}}

```

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

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

```

753 \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.

A	
alias	<u>6</u>
D	
default	<u>6</u>
E	
\ekvcDate	<u>11</u> , <u>6</u> , <u>13</u> , <u>21</u>
\ekvcFlagGetHeight	<u>7</u> , <u>689</u>
\ekvcFlagGetHeights	<u>7</u> , <u>692</u>
\ekvcFlagHeight	<u>7</u> , <u>584</u> , <u>603</u> , <u>633</u> , <u>636</u> , <u>638</u> , <u>690</u> , <u>699</u> , <u>705</u>
\ekvcFlagIf	<u>7</u> , <u>658</u>
\ekvcFlagIfRaised	<u>7</u> , <u>665</u>
\ekvcFlagNew	<u>6</u> , <u>578</u> , <u>592</u> , <u>604</u>
\ekvcFlagRaise	<u>7</u> , <u>600</u> , <u>630</u>
\ekvcFlagReset	<u>7</u> , <u>672</u>
\ekvcFlagSetFalse	<u>7</u> , <u>598</u> , <u>635</u>
\ekvcFlagSetTrue	<u>7</u> , <u>596</u> , <u>635</u>
\ekvcHash	<u>4</u> , <u>308</u>
\ekvcHashAndForward	<u>4</u> , <u>301</u>
\ekvcHashAndUse	<u>4</u> , <u>272</u>
\ekvcSecondaryKeys	<u>5</u> , <u>481</u>
\ekvcSplit	<u>3</u> , <u>91</u>
\ekvcSplitAndForward	<u>3</u> , <u>84</u>
\ekvcSplitAndUse	<u>3</u> , <u>54</u>
\ekvcValue	<u>4</u> , <u>447</u>
\ekvcValueFast	<u>4</u> , <u>454</u>
\ekvcValueSplit	<u>4</u> , <u>455</u>
\ekvcValueSplitFast	<u>5</u> , <u>473</u>
\ekvcVersion	<u>11</u> , <u>6</u> , <u>13</u> , <u>21</u>
\ekvdef	<u>579</u>
\ekvdefNoVal	<u>593</u>
\ekvverr	<u>748</u>
\ekvifdefined	<u>547</u> , <u>563</u>
\ekvifdefinedNoVal	<u>553</u>
\ekvlet	<u>183</u> , <u>369</u> , <u>522</u>
\ekvletkv	<u>549</u>
\ekvletkvNoVal	<u>555</u>
\ekvletNoVal	<u>528</u> , <u>571</u>
\ekvoptarg	<u>9</u>
\ekvoptargTF	<u>9</u>
\ekvparse	<u>136</u> , <u>326</u> , <u>484</u>
\ekvset	<u>42</u> , <u>532</u>
F	
flag-bool	<u>6</u>
flag-false	<u>6</u>
flag-raise	<u>6</u>
flag-true	<u>6</u>
I	
\ifodd	<u>643</u> , <u>650</u> , <u>660</u>
L	
long	<u>5</u>
M	
meta	<u>5</u>
N	
nmeta	<u>5</u>
\number	<u>603</u>
T	
TeX and L ^A T _E X 2 _ε commands:	
\@secondoftwo	<u>38</u> , <u>93</u> , <u>310</u> , <u>663</u> , <u>670</u>
\ekv@alignsafe	<u>50</u>
\ekv@empty	<u>27</u> , <u>28</u> , <u>134</u> , <u>135</u> , <u>141</u> , <u>324</u> , <u>325</u> , <u>330</u> , <u>488</u>
\ekv@endalignsafe	<u>52</u>
\ekv@exparg	<u>593</u> , <u>630</u> , <u>665</u> , <u>748</u>
\ekv@expargtwice	<u>611</u>
\ekv@fi@firstofone	<u>651</u>
\ekv@fi@firstoftwo	<u>36</u> , <u>661</u> , <u>668</u>
\ekv@fi@gobble	<u>33</u> , <u>123</u> , <u>644</u>
\ekv@gobbleto@stop	<u>698</u>
\ekv@ifdefined	<u>148</u> , <u>252</u> , <u>337</u> , <u>380</u> , <u>450</u> , <u>457</u> , <u>495</u> , <u>498</u> , <u>515</u> , <u>581</u>
\ekv@ifempty@	<u>714</u>
\ekv@ifempty@A	<u>710</u> , <u>714</u>
\ekv@ifempty@B	<u>709</u> , <u>710</u>
\ekv@ifempty@false	<u>710</u>
\ekv@name	<u>569</u>
\ekv@stop	<u>584</u> , <u>618</u> , <u>622</u> , <u>626</u> , <u>636</u> , <u>638</u> , <u>641</u> , <u>648</u> , <u>678</u> , <u>681</u> , <u>686</u> , <u>690</u> , <u>691</u> , <u>694</u> , <u>698</u> , <u>699</u> , <u>701</u> , <u>702</u> , <u>704</u> , <u>705</u>
\ekv@zero	<u>396</u> , <u>399</u> , <u>452</u> , <u>477</u>
\ekvc@after@ptype	<u>504</u>
\ekvc@any@long	<u>27</u> , <u>67</u> , <u>76</u> , <u>100</u> , <u>107</u> , <u>134</u> , <u>191</u> , <u>285</u> , <u>293</u> , <u>316</u> , <u>324</u> , <u>375</u>
\ekvc@assert@not@long	<u>526</u> , <u>545</u> , <u>565</u> , <u>577</u> , <u>591</u> , <u>716</u>

\ekvc@defarggobbler	194	\ekvc@long	27 , 141 , 166 , 190 , 330 , 353 , 374 , 488 , 508 , 522 , 716
\ekvc@ekvset@pre@expander	40 , 69 , 78 , 287 , 295	\ekvc@mark	396 , 400 , 407 , 425 , 429 , 437 , 469
\ekvc@ekvset@pre@expander@a	40	\ekvc@p@long	504
\ekvc@ekvset@pre@expander@b	40	\ekvc@safe@found@hash	425 , 465
\ekvc@err	748 , 749 , 750 , 751 , 752	\ekvc@safe@no@hash	429 , 465
\ekvc@err@already@defined	58 , 88 , 95 , 276 , 305 , 312 , 717	\ekvc@safehash@	477
\ekvc@err@empty@hash	477 , 478 , 479 , 480 , 749	\ekvc@safesplithash@	477
\ekvc@err@invalid@bool	586 , 749	\ekvc@set	63 , 69 , 78 , 168 , 171 , 183 , 281 , 287 , 295 , 355 , 358 , 369 , 483 , 521 , 527 , 533 , 547 , 549 , 553 , 555 , 563 , 569 , 571 , 579 , 593 , 746
\ekvc@err@missing@hash .	400 , 429 , 749	\ekvc@setup@hashmacro	370 , 378
\ekvc@err@missing@type .	491 , 511 , 717	\ekvc@setup@splitmacro	185 , 195
\ekvc@err@no@long	716 , 730	\ekvc@SetupHashKeys	282 , 322
\ekvc@err@toomany	103 , 717	\ekvc@SetupHashKeys@a	322
\ekvc@err@unknown@hash .	452 , 459 , 749	\ekvc@SetupHashKeys@b	322
\ekvc@err@unknown@key ..	559 , 573 , 743	\ekvc@SetupHashKeys@c	333 , 339 , 341 , 376
\ekvc@err@unknown@keytype	500 , 517 , 739	\ekvc@SetupSplitKeys	64 , 131
\ekvc@err@value@required	136 , 326 , 484 , 717	\ekvc@SetupSplitKeys@a	131
\ekvc@fasthash@	477	\ekvc@SetupSplitKeys@b	131
\ekvc@fastsplithash@	477	\ekvc@SetupSplitKeys@c	131 , 192
\ekvc@flag@get@height@single ...	689	\ekvc@split@1	195
\ekvc@flag@get@heights	692	\ekvc@split@2	195
\ekvc@flag@get@heights@	692	\ekvc@split@3	195
\ekvc@flag@get@heights@done	692	\ekvc@split@4	195
\ekvc@flag@height	608 , 615	\ekvc@split@5	195
\ekvc@flag@height@	615	\ekvc@split@6	195
\ekvc@flag@name ...	601 , 609 , 612 , 673	\ekvc@split@7	195
\ekvc@flag@namescheme	601 , 613 , 633 , 657 , 667	\ekvc@split@p@long	188
\ekvc@flag@reset	672	\ekvc@stop	42 , 48 , 143 , 146 , 289 , 297 , 317 , 321 , 332 , 335 , 389 , 400 , 408 , 416 , 430 , 438 , 440 , 454 , 465 , 469 , 475 , 478 , 480 , 490 , 493 , 509 , 513
\ekvc@flag@reset@	672	\ekvc@t@alias	543
\ekvc@flag@set@false	635	\ekvc@t@default	561
\ekvc@flag@set@true	635	\ekvc@t@flag-bool	575
\ekvc@hash@p@long	372	\ekvc@t@flag-false	589
\ekvc@hash@pack@argument ..	288 , 321	\ekvc@t@flag-raise	589
\ekvc@helpers@needed	56 , 65 , 86 , 93 , 184 , 274 , 283 , 303 , 310	\ekvc@t@flag-true	589
\ekvc@ifdefined	29 , 57 , 87 , 94 , 275 , 304 , 311	\ekvc@t@meta	519
\ekvc@ifspace ..	142 , 331 , 489 , 506 , 707	\ekvc@t@nmeta	519
\ekvc@ifspace@	707	\ekvc@tmp	2 , 108 , 116 , 117 , 118 , 155 , 158 , 168 , 169 , 182 , 183 , 194 , 196 , 248 , 255 , 268 , 344 , 347 , 355 , 356 , 368 , 369 ,
\ekvc@initials	72 , 80 , 135 , 175 , 177 , 289 , 297 , 325 , 362 , 364		
\ekvc@keycount	26 , 71 , 99 , 102 , 122 , 133 , 140 , 160 , 164 , 179 , 185		

383, 443, 521, 522, 527, 528, 533, 541, 546, 550, 556, 559, 566, 571, 753	
\ekvc@type@flag	<u>589</u>
\ekvc@type@meta	<u>519</u>
\ekvc@type@meta@a	<u>519</u>
\ekvc@type@meta@b	<u>519</u>
\ekvc@undefined	104, 677, 685
\ekvc@zero	<u>26</u>
\ekvcHashAndUse@ ..	277, <u>279</u> , 306, 315
\ekvcSecondaryKeys@a	484, 486
\ekvcSecondaryKeys@b	490, 493
\ekvcSplit@build@argspec ..	106, <u>113</u>
\ekvcSplit@build@argspec@	<u>113</u>
\ekvcSplitAndUse@	59, <u>61</u> , 89, 98
\ekvcValueSplit@recover	<u>455</u>