define expandable \langle key \rangle=\langle value \rangle macros using \texttt{expkv}

Jonathan P. Spratte*

2021-07-07 v1.1

Abstract

\texttt{expkvcs} provides two small interfaces to define expandable \langle key \rangle=\langle value \rangle macros using \texttt{expkv}. It therefore lowers the entrance boundary to expandable \langle key \rangle=\langle value \rangle macros. The stylised name is \texttt{expkvcs} but the files use \texttt{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 4
1.2 Secondary Keys 6
1.2.1 p-type Prefixes 6
1.2.2 t-type Prefixes 6
1.3 Changing the Initial Values 8
1.4 Handling Unknown Keys 9
1.5 Flags 10
1.6 Further Examples 11
1.7 Freedom for Keys! 13
1.8 Speed Considerations 15
1.9 Useless Macros 16
1.10 Bugs 16
1.11 License 16

2 Implementation 17
2.1 The \LaTeX{} Package 17
2.2 The Generic Code 17
2.2.1 Secondary Key Types 33
2.2.2 Flags 38
2.2.3 Helper Macros 41
2.2.4 Assertions 42
2.2.5 Messages 42

\texttt{*jspratte@yahoo.de}
1 Documentation

The \texttt{expkv} package enables the new possibility of creating \texttt{(key)\{value\}} macros which are fully expandable. The creation of such macros is however cumbersome for the average user. \texttt{expkvcs} tries to step in here. It provides interfaces to define \texttt{(key)\{value\}} macros without worrying too much about the implementation. In case you’re wondering now, the \texttt{cs} in \texttt{expkvcs} stands for control sequence, because \texttt{def} was already taken by \texttt{expkvdef} and “control sequence” is the term D. E. Knuth used in his \TeX{}book for named commands hence macros (though he also used the term “macro”). So \texttt{expkvcs} defines control sequences for and with \texttt{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 \texttt{(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 \texttt{expkvcs}. 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 \texttt{expkv} or using \texttt{expkvdef} for convenience. Or you resort to another \texttt{(key)\{value\}} interface. Nevertheless setting up macros with \texttt{expkvcs}, especially with \texttt{\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.

\texttt{expkvcs} is usable as generic code and as a \LaTeX{} package. It’ll automatically load \texttt{expkv} in the same mode as well. To use it, just use one of

\begin{verbatim}
\usepackage{expkv-cs} % LaTeX
\input expkv-cs % plainTeX
\end{verbatim}

1.1 Define Macros and Primary Keys

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

To allow keys not defined \texttt{long} which key names should start with \texttt{long}, you can also use the prefix \texttt{short} (\texttt{short} and \texttt{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 \texttt{short}'s existance, essentially it does nothing.

The consequence culminates in the following:

\begin{verbatim}
\ekvcSplit \foo \\
\{ \\
\indent long short = abc\par \\
\indent ,short long = def \\
\} \\
\{#1#2\}
\end{verbatim}

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

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

At the moment \texttt{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 \texttt{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.

\begin{verbatim}
\ekvcSplit \foo \\
\{ \\
\indent \langle key\rangle=\langle value\rangle \\
\} \\
\{#1#2\}
\end{verbatim}

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

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

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

\begin{verbatim}
a is a.
b is b.
a is a.
b is e.
\end{verbatim}
ekvcSplitAndForward \ekvcSplitAndForward{cs}{(after)}{(primary keys)}

This defines \texttt{cs} to be a macro taking one mandatory argument which should contain a \texttt{key}=(\texttt{value}) list. You can use as many primary keys as you want with this. The primary keys will be forwarded to \texttt{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 \texttt{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 \texttt{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, \texttt{cs} will use what follows the \texttt{key}=(\texttt{value}) list. So its syntax will be

\texttt{cs}\{\texttt{key}=(\texttt{value}), \ldots\}\{\texttt{after}\}

and the code in \texttt{after} should expect at least as many arguments as the number of keys defined for \texttt{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 \texttt{ekvcValue}, not the parsing, and for a key access using \texttt{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 \texttt{cs} to be a macro taking one mandatory argument which should contain a \texttt{key}=(\texttt{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 \texttt{definition}, in which you can access the \texttt{value} of a \texttt{key} by using \texttt{ekvcValue\{key\}}\{#1\}.

Example: This defines an equivalent macro to the \texttt{\foo} defined with \texttt{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. |
\texttt{ekvcHashAndForward} \texttt{ekvcHashAndForward}((cs){(after)}{(primary keys)})

This defines \texttt{(cs)} to be a macro taking one mandatory argument which should contain a \texttt{(key)=\langle value \rangle} 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 \texttt{(after)}. You can use a single macro for \texttt{(after)} or use some arbitrary stuff, which will be left in the input stream before the hashed \texttt{(key)=\langle value \rangle} list with one set of braces stripped. In the macro called in \texttt{(after)} you can access the \texttt{(value)} of a \texttt{(key)} by using \texttt{ekvcValue{(key)}{\#1}} (or whichever argument the hashed \texttt{(key)=\langle value \rangle} list will be).

\textit{Example:} This defines a macro \texttt{\foo} processing two keys, and passing the result to \texttt{\foobar}:

\begin{verbatim}
\ekvcHashAndForward \foo \foobar \{a=a, b=b\}
\end{verbatim}

\begin{verbatim}
newcommand* \foobar[1]{a is \ekvcValue{a}{\#1}. \par
b is \ekvcValue{b}{\#1}. \par}
\end{verbatim}

\texttt{ekvcHashAndUse} \texttt{ekvcHashAndUse}((cs){(primary keys)})

This will roughly do the same as \texttt{ekvcHashAndForward}, but instead of specifying what will be used after hashing the keys at install time, \texttt{(cs)} will use what follows the \texttt{(key)=\langle value \rangle} list. So its syntax will be

\texttt{(cs){\langle key \rangle=\langle value \rangle, ...}{(after)}}

\texttt{ekvcValue} \texttt{ekvcValue}((key){(key list)})

This is a safe way to access your keys in a hash variant. \texttt{(key)} is the key which’s \texttt{(value)} you want to use out of the \texttt{(key list)}. \texttt{(key list)} should be the key list argument forwarded to your underlying macro by \texttt{ekvcHash}, \texttt{ekvcHashAndForward}, or \texttt{ekvcHashAndUse}. It will be tested whether the hash function to access that \texttt{(key)} exists, the \texttt{(key)} argument is not empty, and that the \texttt{(key list)} really contains a \texttt{(value)} of that \texttt{(key)}. This macro needs exactly two steps of expansion and if used inside of an \texttt{edef} or \texttt{expanded} context will protect the \texttt{(value)} from further expanding.

\texttt{ekvcValueFast} \texttt{ekvcValueFast}((key){(key list)})

This behaves just like \texttt{ekvcValue}, but \textit{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 \texttt{(key)} isn’t part of the \texttt{(key list)} (e.g., because it was defined as a key for another macro – all macros share the same hash function per \texttt{(key)} name). Use it if you know what you’re doing. This macro needs exactly three steps of expansion in the no-errors case.

\texttt{ekvcValueSplit} \texttt{ekvcValueSplit}((key){(key list)}{(next)})

If you need a specific \texttt{(key)} from a \texttt{(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 \texttt{ekvcValueSplit} will extract one specific \texttt{(key)}’s value from the list and forward it as an argument to \texttt{(next)}, so the result of this will be \texttt{(next){\langle value \rangle}}. This is roughly as fast as \texttt{ekvcValue} and runs the same tests.

\textit{Example:} The following defines a macro \texttt{\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:

\texttt{\textbackslash ekvcHash \textbackslash foo \{a=a, b=b, c=c\} \textbackslash \texttt{\textbackslash ekvcValueSplit}\{a\}\{#1\}\texttt{\textbackslash fooBar\{#1\}}}

\texttt{\texttt{\textbackslash newcommand}\texttt{\textbackslash fooBar\{2\}\{a is \#1. \texttt{\textbackslash par} b is \texttt{\textbackslash ekvcValue\{b\}\{#2\}. \texttt{\textbackslash par} c is \texttt{\textbackslash ekvcValue\{c\}\{#2\}. \texttt{\textbackslash par}}}}

\texttt{\textbackslash foo{}}

This behaves just like \texttt{\textbackslash ekvcValueSplit}, but it won’t run the same tests, hence it is faster but more error prone, just like the relation between \texttt{\textbackslash ekvcValue} and \texttt{\textbackslash ekvcValueFast}.

\subsection{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 \texttt{long}) which are called p-type prefix and must have a type (like \texttt{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|.

\texttt{\texttt{\textbackslash ekvcValueSplitFast\{\textbackslash textbackslash key\}\{\textbackslash textbackslash key\ list\}\{\textbackslash textbackslash next\}}} This behaves just like \texttt{\textbackslash ekvcValueSplit}, but it won’t run the same tests, hence it is faster but more error prone, just like the relation between \texttt{\textbackslash ekvcValue} and \texttt{\textbackslash ekvcValueFast}.

\subsection{p-type Prefixes}

There is only one p-prefix available, which is \texttt{long}.

\subsection{t-type Prefixes}

If you’re familiar with \texttt{expVal} 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 (\texttt{long}), which will be available in grey (\texttt{long}), and which will be disallowed in red (\texttt{long}). This will be put flush right next to the syntax line.
With a meta key you can set other keys. Whenever \langle key\rangle is used the keys in the \langle key\rangle=\langle value\rangle list will be set to the values given there. You can use the \langle value\rangle given to \langle key\rangle by using #1 in the \langle key\rangle=\langle value\rangle list. The keys in the \langle key\rangle=\langle value\rangle list can be primary and secondary ones.

An nmeta key is like a meta key, but it doesn’t take a value, so the \langle key\rangle=\langle value\rangle list is static.

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 expkv 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.

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 \ekvcSplit and friends (see \ekvcChange in subsection 1.3 for this). \langle key\rangle can be a primary or secondary key.

While all other key types replace the current value of the associated \langle primary\rangle 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 \langle primary\rangle should be the name of a \langle primary\rangle key, and \langle definition\rangle the way you want to store the current and the new value. Inside \langle definition\rangle you can use #1 for the current, and #2 for the new value. The \langle definition\rangle will not expand any further during the entire parsing (so this doesn’t allow general processing of current and new values). The resulting \langle key\rangle will inherit being either short or long from the \langle primary\rangle 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}
```

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

This is a secondary key that doesn't directly involve any of the primary or secondary keys. This defines \( \langle \text{key} \rangle \) to take no value and set the flag called \( \langle \text{cs} \rangle \) to true or false, respectively. If \( \langle \text{cs} \rangle \) isn't defined yet it will be initialised as a flag. Please also read subsection 1.5.

This is a secondary key that doesn't directly involve any of the primary or secondary keys. This defines \( \langle \text{key} \rangle \) to take no value and raise the flag called \( \langle \text{cs} \rangle \). If \( \langle \text{cs} \rangle \) isn't defined yet it will be initialised as a flag. Please also read subsection 1.5.

1.3 Changing the Initial Values

This processes the \( \langle \text{key} \rangle \)=\langle \text{value} \rangle \) list for the macro \( \langle \text{cs} \rangle \) 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). \( \langle \text{cs} \rangle \) should be defined with \texttt{expkvcs} (so with any of the methods in subsection 1.1). Inside the \( \langle \text{key} \rangle \)=\langle \text{value} \rangle \) list both primary and secondary keys can be used. If \( \langle \text{cs} \rangle \) was defined \texttt{long} earlier it will still be \texttt{long}, every other \TeX{} prefix will be stripped (but \texttt{expkvcs} doesn’t support them anywhere else so that should be fine). The resulting new defaults will be stored inside the \( \langle \text{cs} \rangle \) locally (just as the original defaults were). If there was an unknown key forwarding added to \( \langle \text{cs} \rangle \) (see subsection 1.4) any unknown key will be stored inside the list of unknown keys as well. \texttt{ekvcChange} is not expandable!

Consider the following example:

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

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 \texttt{\foo}, hence this might not be as powerful as similar macros used with other \( \langle \text{key} \rangle \)=\langle \text{value} \rangle \) 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 \ldots 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 \{\textlangle\textrangle\}, so with spaces around one set of braces (this way, most other \textlangle\textrangle 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 \ldots 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 \ldots at the same time.

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

\begin{verbatim}
\newcommand*{\foo}{\ekvoptarg\fooKV{}}
\ekvcSplitAndForward\fooKV\fooOUT{{}}

\newcommand{\fooOUT}[5]{{}
\a=a
\ldots
\b=b
\ldots={}
}\end{verbatim}

\begin{verbatim}
\newcommand\fooOUT[5]{{}
\% 
a is #1 and b is #3.\par
\includegraphics{{#2}|{#5}}\par
\texttt{\ldots} is #4.\par
}
\foo[width=.5\linewidth, b=c, \ldots={\{a stupid key name, but works\}}]
\end{verbatim}

Inside the hash variants the unknown keys list will be put inside the hash named \ldots (we have to use some name, so why not this). As a consequence a primary key named \ldots 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 \ldots key. We have to make sure that \includegraphics sees the *\langle key\rangle=\langle value\rangle* list, so need to expand \ekvcValue{\ldots}\{\#1\} before \includegraphics parses it.
\newcommand{\foo}{\ekvoptarg{\fooKV}{}}
\ekvHashAndForward{\fooKV}{\fooOUT}
\{a=a, \ b=b, \ldots\}
\newcommand{\fooOUT}[2]{%  
    a \text{ is } \ekvcValue{a}{\#1} \text{ and}
    b \text{ is } \ekvcValue{b}{\#1}. \par
    \expanded{\noexpand\includegraphics{\ekvcValue{\ldots}{\#1}}}
    \#2\par
  }
\foo[width=\linewidth, \ b=c]
{example−image−duck−portrait}

1.5 Flags

The idea of flags is taken from \expl. 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!

\exPkvcs provides some macros to access, alter, and use flags. Flags of \exPkvcs don’t share a name space with the flags of \expl.

\ekvcFlagNew{\langle\text{flag}\rangle}
This initialises the macro \langle\text{flag}\rangle as a new flag. It isn’t checked whether the macro \langle\text{flag}\rangle is currently undefined. A \langle\text{flag}\rangle 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 \langle\text{flag}\rangle with \ekvcFlagNew.

\ekvcFlagHeight{\langle\text{flag}\rangle}
This expands to the current height of \langle\text{flag}\rangle in a single step of expansion (without a trailing space).

\ekvcFlagRaise{\langle\text{flag}\rangle}
This expandably raises the height of \langle\text{flag}\rangle by 1.

\ekvcFlagSetTrue{\langle\text{flag}\rangle}
\ekvcFlagSetFalse{\langle\text{flag}\rangle}
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\text{flag}\rangle to true or false, respectively, by raising it if necessary.
This interprets a ⟨flag⟩ as a boolean and expands to either ⟨true⟩ or ⟨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⟩.

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.

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

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 ⟨flags⟩. 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).

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

#1 is a #5 from #3. #6 is of #2 age and #7 hobby is #4.\par

\end{verbatim}

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

\ekvcSecondaryKeys \character
  |
  alias pro = pronoun,
\textit{alias} \texttt{pos = possessive},
\texttt{nmeta me =}
\begin{verbatim}
{ name=Jonathan,
  age=a young,
  nationality=Germany,
  hobby=\TeX\ coding,}
\end{verbatim}
\texttt{meta lady =}
\begin{verbatim}
{ type=Lady, pronoun=She, possessive=her, name=Jane Doe, #1},
\end{verbatim}
\texttt{nmeta paulo =}
\begin{verbatim}
{ name=Paulo,
  type=duck,
  age=a young,
  nationality=Brazil,
  hobby=to quack,}
\end{verbatim}

Now we can describe people using
\begin{verbatim}
\character{ }
\character{me}
\character{paulo}
\character{lady={name=Evelyn, nationality=Ireland, age=the best, hobby=reading}}
\character{}
\end{verbatim}

As one might see, the \texttt{lady} key could actually have been an \texttt{nmeta} key as well, as all that is done with the argument is using it as a \langle\texttt{key}\rangle=\langle\texttt{value}\rangle 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 \texttt{xparse}'s \texttt{\ekvoptarg} or \texttt{\ekvoptargTF} and forwarding arguments one can easily define \langle\texttt{key}\rangle=\langle\texttt{value}\rangle macros with actual optional and mandatory arguments as well. A small nonsense example (which should perhaps use \texttt{\ekvcSplitAndForward} instead of \texttt{\ekvcHashAndForward} since it only uses four keys and one other argument – and isn’t expandable since it uses a \texttt{tabular} environment, so it would’ve been better to use a more feature rich \langle\texttt{key}\rangle=\langle\texttt{value}\rangle interface most likely, e.g., the one provided by \texttt{\ekvDef}):

\begin{verbatim}
\makeatletter
\newcommand*{\nonsense}{\ekvoptarg\nonsense@a{}}
\end{verbatim}
\begin{verbatim}
\ekvcHashAndForward\nonsense@a\nonsense@b
\end{verbatim}

12
\begin{tabular}{llll}
key & A & \ekvcValue{keyA}{#1} & \\
& B & \ekvcValue{keyB}{#1} & \\
& C & \ekvcValue{keyC}{#1} & \\
& D & \ekvcValue{keyD}{#1} & \\
\multicolumn{2}{l}{mandatory} & #2 & \\
\end{tabular}

And then we would be able to do some nonsense
\nonsense
\nonsense\[\text{keyA=hihi}\]{haha}
\nonsense\[\text{keyA=hihi, keyB=A}\]{hehe}
\nonsense\[\text{keyC=huhu, keyA=hihi, keyB=A}\]{haha}

resulting in

\begin{tabular}{llllllllll}
key & A & A & key & A & hihi & key & A & hihi & key & A & hihi \\
C & c & C & c & C & c & C & c & c & c & c \\
D & d & D & d & D & d & D & d & D & d & D \\
mandatory & mandatory & haha & mandatory & hehe & mandatory & haha & \\
\end{tabular}

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 \expkvcs’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 \expkvcs by using the low-level interface of \expkv or even the interface provided by \expkvdef. The set name used for \expkvcs’s keys is the macro name, including the leading backslash, or more precisely $\texttt{\string}⟨\texttt{cs}⟩$ 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:

\[
ekvcPass \langle \text{cs} \rangle \{ \langle \text{key} \rangle \} \{ \langle \text{value} \rangle \}
\]

This passes \langle value \rangle on to \langle key \rangle for the \texttt{expvc}-macro \langle cs \rangle. It should be used inside the key parsing of a macro defined with \texttt{expvc}, 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:

\[
\texttt{\ekvcSplit \foo \{a=A, b=B\}\{a is \ #1. \par b is \ #2. \par\}}
\]
\[
\texttt{\ekvcSecondaryKeys \foo \{a\}\{#1\} \texttt{\ekvcPass \foo \{b\}\{#2\}}}
\]
\[
\texttt{\foo\{\}}
\]
\[
\texttt{\foo\{c=1-2\}}
\]

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

\[
\texttt{process} \langle \text{key} \rangle = \{ \langle \text{primary} \rangle \} \{ \langle \text{definition} \rangle \}
\]

This will grab the current value of a \langle primary \rangle key as \#1 (without changing the current value) and the new value as \#2 and leave all the processing to \langle definition \rangle. You should use \texttt{ekvcPass} to forward the values afterwards. Unlike aggregate you can specify whether the \langle key \rangle should be long or not, this isn’t inherited from the \langle primary \rangle 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:

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

\(^{1}\text{This means that the } \langle \text{key}\rangle=\langle \text{value}\rangle\text{-list can’t contain alignment markers that are not inside an additional set of braces if used inside a \TeX alignment.}\)
1.8 Speed Considerations

As already mentioned in the introduction there are some speed considerations implied if you choose to define macros via \texttt{expkv}. However the overhead isn’t the factor which should hinder you to use \texttt{expkv} 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 \texttt{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 \texttt{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 \texttt{expkv} with the \texttt{expkv-def} definitions, namely the split example from above, a hash variant using \texttt{\ekvcValue} and a hash variant using \texttt{\ekvcValueFast}.

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

The first comparison removes the typesetting part from all the definitions, so that only the key parsing is compared. In this comparison the \texttt{\ekvcValue} and \texttt{\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 \texttt{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 \texttt{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 \texttt{expkvdef}.

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 \texttt{(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 \texttt{(key)=(value)} parsing macros you should go on and define them using \texttt{expkvcs}. If you just want to define a simple macro with a few keys \texttt{\ekvcSplit} might be the easiest interface there is. If you have a reasonable use case for \texttt{(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 \texttt{(key)=(value)} setups (but if you’re after maximum speed there aren’t that many \texttt{(key)=(value)} parsers beating \texttt{expkvcs}). And if you are after maximum performance maybe ditching the \texttt{(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.

\begin{tabular}{l}
\texttt{\ekvcDate} \\
\texttt{\ekvcVersion}
\end{tabular}

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: \url{https://github.com/Skillmon/tex_expkv-cs}

1.11 License

Copyright © 2020–2021 Jonathan P. Spratte

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

\url{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 \LaTeX Package

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

\begin{verbatim}
\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%
}
\end{verbatim}

\texttt{\input\texttt{expkv-cs.tex}}

We make sure that $\texttt{expkv-cs.tex}$ is only input once:

\begin{verbatim}
\csname ekvcVersion\endcsname
\else
\expandafter\endinput
\fi
\end{verbatim}

\texttt{\ekvcVersion} \texttt{\ekvcDate}

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

\begin{verbatim}
\def\ekvcVersion{1.1}
\def\ekvcDate{2021-07-07}
\end{verbatim}

(End definition for \texttt{\ekvcVersion} and \texttt{\ekvcDate}. These functions are documented on page 16.)

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 \texttt{\ekvc@tmp} to use \texttt{\ProvidesFile}, else this will expand to a \texttt{\relax} and do no harm.

\begin{verbatim}
\csname ekvc@tmp\endcsname
\end{verbatim}

Store the category code of \texttt{@} to later be able to reset it and change it to \texttt{11} for now.

\begin{verbatim}
\expandafter\chardef\csname ekvc@tmp\endcsname=\catcode'@\0
\catcode'@=11
\end{verbatim}

\texttt{\ekvc@tmp} will be reused later, but we don't need it to ever store information long-term after $\texttt{expkvcs}$ 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
      \@firstoftwo
      \ekv@fi@firstoftwo
      \@secondoftwo
    \fi
  }
  (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 \expkvcs 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 \expkvcs 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\ekvset{#1}\ekvc@stop\ekvc@stop
  }
  \def\ekvc@ekvset@pre@expander@b#1\ekvc@stop#2\ekvc@stop{%
    \unexpanded\expandafter{\ekv@alignsafe}\%\unexpanded{#1}\unexpanded{#2}\%
    \unexpanded\expandafter{\ekv@endalignsafe}\%
  }
  (End definition for \ekvc@ekvset@pre@expander.)
\texttt{\textbackslash ekvc\textbackslash SplitAndUse} \texttt{\textbackslash ekvc\textbackslash SplitAndUse@}

The first user macro we want to set up can be reused for \texttt{\textbackslash ekvc\textbackslash SplitAndForward} and \texttt{\textbackslash ekvc\textbackslash Split}. We'll split this one up so that the test whether the macro is already defined doesn't run twice.

\begin{verbatim}
\protected\long\def\ekvcSplitAndUse#1#2{% 
  \let\ekvc@helpers@needed@firstoftwo 
  \ekvc@ifdefined#1{% \ekvc@err@already@defined#1}% 
  {\ekvcSplitAndUse@#1{}{#2}}% 
}%
\end{verbatim}

\textit{(End definition for \texttt{\textbackslash ekvc\textbackslash SplitAndUse}. This function is documented on page 4.)}

\texttt{\textbackslash ekvc\textbackslash SplitAndUse@}

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

\begin{verbatim}
\protected\long\def\ekvcSplitAndUse@#1#2#3{% 
  \edef\ekvc@set{\string#1}% 
  \ekvc\textbackslash SetupSplitKeys{#3}% 
  \ekvc@helpers@needed 
  {% \ekvc@any@long\edef#1##1{% 
    \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}% 
    \unexpanded\expandafter{\csname ekvc@split@\the\ekvc@keycount\endcsname}% 
    \unexpanded\expandafter{\ekvc@initials{}#2}% 
  }% 
  }% 
  {% \ekvc@any@long\edef#1##1{% 
    \expandafter\ekvc@ekvset@pre@expander\expandafter{\ekvc@set}% 
    \unexpanded\expandafter{\ekvc@initials}% 
  }% 
}%
\end{verbatim}

\textit{(End definition for \texttt{\textbackslash ekvc\textbackslash SplitAndUse@})}

\texttt{\textbackslash ekvc\textbackslash SplitAndForward}

This just reuses \texttt{\textbackslash ekvc\textbackslash SplitAndUse@} with a non-empty second argument, resulting in that argument to be called after the splitting.

\begin{verbatim}
\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{verbatim}
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.

```latex
\begin{verbatim}
\protected\long\def\ekvcSplit#1#2#3\
{\let\ekvc@helpers@needed\@secondoftwo
 \ekvc@iifdefined#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\endcsname{#3}
 \fi
 \fi}
\end{verbatim}
```

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

```latex
\begin{verbatim}
\protected\def\ekvcSetupSplitKeys\
{\ekvc@SetupSplitKeys\ekvc@SetupSplitKeys@@\ekvc@SetupSplitKeys@@b\ekvc@SetupSplitKeys@c\ekvc@SetupSplitKeys@back@unknown}
\end{verbatim}
```
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`.

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.

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`.

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.
If no value was provided this could either be an error, or the unknown key forwarding. We have to check this (comparing against \ekvc@tripledots 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.

The \begingroup\expandafter\endgroup ensures that the split mark isn't actually defined (even if it just were with meaning \relax).
The long prefix lets the internals \ekvc@long and \ekvc@any@long to \long so that the key macro will be long.

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

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@c}

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}}

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\unexpanded\expandafter{\csname ekvc@split@1\endcsname}\
\unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1##2##3\
{##3{##1}##2}\
\long\gdef\unexpanded\expandafter{\csname ekvc@split@2\endcsname}\
\unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1\
\unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2\
##3##4\
{##4{##1}{##2}##3}\
\long\gdef\unexpanded\expandafter{\csname ekvc@split@3\endcsname}\
\unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1\
\unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2\
\unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3\
##4##5\
{##5{##1}{##2}{##3}##4}\
\long\gdef\unexpanded\expandafter{\csname ekvc@split@4\endcsname}\
\unexpanded\expandafter{\csname ekvc@splitmark@1\endcsname}##1\
\unexpanded\expandafter{\csname ekvc@splitmark@2\endcsname}##2\
\unexpanded\expandafter{\csname ekvc@splitmark@3\endcsname}##3\
\unexpanded\expandafter{\csname ekvc@splitmark@4\endcsname}##4\
##5##6\
{##6{##1}{##2}{##3}{##4}##5}\
\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}\
      \unexpanded{\ekvc@hash@pack@argument}\
      \unexpanded\expandafter{\ekvc@initials\ekvc@stop#2}%
    }%
  }%
}

(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%
{\
}

25
\let\ekvc@helpers@needed\@secdotwo
\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@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@any@space\ekv@empty
 \ekvparse\ekvc@SetupHashKeys@check@unknown\ekvc@SetupHashKeys@\#1
}
\protected\def\ekvc@SetupHashKeys@\#1
{\%
 \let\ekvc@any@long\ekv@empty
 \ekvc@ifspace\#1
 {\ekvc@SetupHashKeys@\#1\ekvc@stop}
 {\ekvc@SetupHashKeys@\#1}
}
\protected\def\ekvc@SetupHashKeys@\#1 #2\ekvc@stop
{\%
 \ekvc@ifdefined\ekvc@Hash@p#1
 {\csname ekvc@Hash@p#1\endcsname\#2}
 {\ekvc@SetupHashKeys@\#1 \#2}
}

Yes, even the defining macro looks awfully familiar. Instead of numbered we have named marks. Still the key macros grab everything up to their respective mark and reorder the arguments. The same quirk is applied for short keys. And instead of the \ekvc@setup@splitmacro we use \ekvc@setup@hashmacro.
\protected\long\def\ekvc@SetupHashKeys@\#1
{\%
 \begingroup
 \edef\ekvc@tmp
 {\endgroup
 \long\def\unexpanded{\ekvc@tmp}####1####2
 \unexpanded\expandafter{\csname ekvc@Hashmark@#1\endcsname}####3}
}

26
(End definition for \texttt{\% \ekvc@SetupHashKeys} and \texttt{others}.)

27
Nothing astonishing here either.

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

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@mark 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.


\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% {
\unexpanded{\ekv@ifdefined{ekvc@safehash@#1}{\csname ekvc@safehash@#1\endcsname}{\ekvc@err@unknown@hash{#1}@firstoftwo{}}}%
}(End definition for \ekvcValue. This function is documented on page 5.)
To be as fast as possible, this doesn’t test for anything, assuming the user knows best.

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

(END definition for \ekvcValueFast. This function is documented on page 5.)

This splits off a single value.

\long\def\ekvcValueSplit#1\%
\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 and \ekvcValueSplit@recover. These functions are documented on page 5.)

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

\long\def\ekvcValueSplitFast#1#2\csname ekvc@fastsplithash@#1\endcsname#2\ekvc@stop

(END definition for \ekvcValueSplitFast. This function is documented on page 6.)

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@fastsplithash@#1\ekvc@stop{\ekvc@err@empty@hash#2{}}

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

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

\protected\long\def\ekvcSecondaryKeys#1#2\%
\edef\ekvc@set{\string#1}\%
\ekvparse\ekvc@err@value@required\ekvcSecondaryKeys@a{#2}\

\protected\def\ekvcSecondaryKeys@a#1\%
\let\ekvc@long\ekv@empty

30
This can be used to change the defaults of an `expkvcs` 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`.

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.

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.

Next we expand the macro once to get its contents (including the current default values with their markers). #1 is either `\long` or empty, #2 is the macro.
Here we place an unbalanced closing brace after the expansion of the macro. Then we just parse the \{key\}=\{value\}-list with \ekvs set, that will exchange the values behind the markers. Once those are changed we give control to \ekv c@change@d. The \ekvs et 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.

\[\text{579 } \text{\ekv@exparg{\texttt{\protect\long\def\ekvc@change@cc#1#2#3#4}}}}\%\]
\[\text{580 } \{\%\}
\[\text{581 } \text{\expandafter\iffalse\expandafter{\expandafter\fi}
\text{\ekvs et{\string#3}{#4}}}\%\]
\[\text{582 } \text{\ekvc@change@dd\{#2\}{#3}}}\%
\[\text{583 } \text{#1}}}\%
\[\text{584 } \}%\]
\[\text{585 } \}%\]
\[\text{586 } \} %\]

The final step needs to put an unbalanced opening brace after \def. We do that with the help of a temporary macro which stores everything necessary for \def and expand an \iffalse\fi construct to leave a single opening brace. #1 will be either empty or \long and #2 is the macro. Each of the macros defined with \exp kc s takes exactly one parameter, so we put that here as \###1.

\[\text{587 } \text{\protect\def\ekvc@change@dd#1#2#3#4\%}\]
\[\text{588 } \{\%
\[\text{589 } \text{\def\ekvc@tmp\{#1\def#2\###1\}}\%
\[\text{590 } \text{\expandafter\ekvc@tmp\expandafter{\iffalse\fi}}\]
\[\text{591 } \} %\]

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

\(\text{\ekvc@change@iflong}\) \(\text{\ekvc@change@iflong@}\)

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.

\[\text{592 } \text{\def\ekvc@change@iflong#1\%}\]
\[\text{593 } \{\%
\[\text{594 } \text{\protected\def\ekvc@change@iflong\###1\%}\]
\[\text{595 } \{\text{\expandafter\ekvc@empty\expandafter{\ekvc@change@iflong\###1\}}\%
\[\text{596 } \text{\def\ekvc@change@iflong\###1\}}\%
\[\text{597 } \} %\]
\[\text{598 } \text{\expandafter\ekvc@change@iflong\expandafter{\text{\string\long}}\} %\]

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

\(\text{\ekvcPass}\)

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 implementation is pretty straightforward.

\[\text{599 } \text{\long\def\ekvcPass#1#2\%}\]
\[\text{600 } \{\%
\[\text{601 } \text{\ekvifdef\texttt{\string#1}{\#2}\%\}
\[\text{602 } \{\text{\csname\ekv@name{\string#1}{\#2}\endcsname}\%
\[\text{603 } \{\ekvc@err\texttt{unknown key or macro\{#1\}{\#2}\}@gobble\%\}
\[\text{604 } \} %\]

(End definition for \ekvcPass. This function is documented on page 14.)
2.2.1 Secondary Key Types

The prefixes are pretty straightforward again. Just set \texttt{\ekvc@p@long} and \texttt{\ekvc@after@ptype}.

\begin{verbatim}
\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
  {
    \ifdefined{ekvc@t@#1}
    {\csname ekvc@t@#1\endcsname{#2}}
  \else
    \ekvc@err@unknown@keytype{#1}\@gobble%
  \fi
  }

  (End definition for \texttt{\ekvc@p@long} and \texttt{\ekvc@after@ptype}.)
\end{verbatim}

\texttt{\ekvc@t@meta} \texttt{\ekvc@nmeta} \texttt{\ekvc@type@meta} \texttt{\ekvc@type@meta@a} \texttt{\ekvc@type@meta@b} The meta and nmeta key types use a nested \texttt{\ekvset} to set other keys in the same macro's \texttt{\set{}}.

\begin{verbatim}
\protected\def\ekvc@t@meta
  {\edef\ekvc@tmp{\ekvc@set}\expandafter\ekvc@type@meta\expandafter{\ekvc@tmp}\ekvc@long{##1}\ekvlet}
\protected\def\ekvc@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}{#5}\ekvc@tmp}
\protected\def\ekvc@type@meta@a
  {\expandafter\ekvc@type@meta@b#2#3}
\protected\def\ekvc@type@meta@b#1#2#3%
  {#2\def\ekvc@tmp#3(#1)}

(End definition for \texttt{\ekvc@t@meta} and others.)
\end{verbatim}

\texttt{\ekvc@t@alias} \texttt{\ekvc@type@meta} alias just checks whether there is a key and/or \texttt{NoVal} key defined with the target name and \texttt{\let} the key to those.

\begin{verbatim}
\protected\def\ekvc@t@alias#1#2%
\end{verbatim}
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\{
  \ekvc@assert@not@long{default #1}\%
  \edef\ekvc@tmp\unexpanded\expandafter{%\csname\ekv@name\ekvc@set{#1}\endcsname{#2}}%
  \ekvc@letNoVal\ekvc@set{#1}\ekvc@tmp
\}
```

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

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\ekvc@type@aggregate\ekvc@type@aggregate@long\ekvc@type@aggregate@short\{process\}%
\}
```

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

```latex
\texttt{\detokenize{\begin{longdef}{ekvc@type@aggregate}{\#1\#2\#3\#4\#5}
{\%}
{ekvc@assert@twoargs{\#5}{\#3 \#4}{ekvc@type@aggregate@a\#1\{\#4\}\#5}}
{\%}
\end{longdef}}}
\texttt{\detokenize{\begin{longdef}{ekvc@type@aggregate@b}{\#1\#2\#3\#4\#5}
{\%}
ekvifdefined{ekvc@set}{\#4}
{\%}
{\texttt{\begin{longdef}{ekvc@type@aggregate@check@long}{\#1\#2}{\#3}}}
{\texttt{ekvc@type@aggregate@check@long@b}}
\end{longdef}}}
\texttt{\detokenize{\begin{longdef}{ekvc@type@aggregate@long}{\#1\#2\#3}}}
{\texttt{\detokenize{ekvc@type@aggregate@long@0}}}
```

To check whether the primary key is long we see whether its \texttt{\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 \texttt{\detokenize}), and afterwards get the \texttt{\meaning} of the macro. A temporary helper does the real test by gobbling and forwarding the resulting to \texttt{ekvc@ifempty}.

```latex
\texttt{\begin{longdef}{ekvc@type@aggregate@check@long}{\#1\#2}{\#3}}
{\%}
ekvc@type@aggregate@check@long@b
{\texttt{ekvc@type@aggregate@check@long@0}}
{\texttt{\endcsname}}
{\%}
{\texttt{\endgroup}}
{\texttt{\begingroup}}
{\texttt{\expandafter\endgroup\expandafter ekvc@type@aggregate@check@long@b}}
{\texttt{\csname ekvc@extract@mark#1\endcsname}}
{\%}
{\texttt{\endgroup}}
```

(\textit{End definition for ekvc@type@aggregate, ekvc@type@aggregate@a, and ekvc@type@aggregate@b.})

The long variant just builds the split mark we extract, uses the hack to swap argument order, and then does the definition via \texttt{ekvc@let} and a temporary macro.

```latex
\texttt{\begin{longdef}{ekvc@type@aggregate@long}{\#1}}
{\%}
{\texttt{\begingroup}}
{\texttt{\expandafter\endgroup}}
{\texttt{\expandafter ekvc@type@aggregate@long@0}}
{\texttt{\expandafter ekvc@type@aggregate@long@0}}
{\texttt{\expandafter ekvc@type@aggregate@long@0}}
{\texttt{\expandafter ekvc@type@aggregate@long@0}}
```

(\textit{End definition for ekvc@type@aggregate@check@long, ekvc@type@aggregate@check@long@0, ekvc@type@aggregate@check@long@1, ekvc@type@aggregate@check@long@2, ekvc@type@aggregate@check@long@3, and ekvc@type@aggregate@check@long@4.})
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.

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

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.
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.

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

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

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

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

\ekvc@flag@name

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

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

(End definition for \texttt{\ekvc@flag@name} and \texttt{\ekvc@flag@namescheme}.)

\ekvcFlagHeight

For semantic reasons we use \texttt{\number} with another name.

\let\ekvcFlagHeight\number

(End definition for \texttt{\ekvcFlagHeight}. This function is documented on page 10.)

\ekvcFlagNew

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

\protected\def\ekvcFlagNew#1%

{%
    \edef#1%
    {%
        \unexpanded{\ekvc@flag@height}%
        \unexpanded{\expandafter{\csname\ekvc@flag@name\#1\endcsname}}%
    }%
\expandafter\def\csname\ekvc@flag@name#1\endcsname##1%

{\expandafter\ekvc@flag@namescheme\expandafter{\string#1}{##1}}%
}

(End definition for \texttt{\ekvcFlagNew}. This function is documented on page 10.)

\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 \texttt{\ekvc@flag@height} ends the \texttt{\number} evaluation.

\def\ekvc@flag@height#1%

{%}
Raising a flag simply means letting the `\ekvcFlagSetTrue` macro for the current height to relax. The result of raising a flag is that its height is bigger by 1.

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.

We can expand `\ekvcFlag NAMESCHEMEl` at definition time here, which is why we're using a temporary definition to set up `\ekvcFlagSetTrue` and `\ekvcFlagSetFalse`.

(End definition for `\ekvcFlagHeight` and `\ekvcFlagHeight@`.)

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

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.

We can expand `\ekvcFlag NAMESCHEMEl` at definition time here, which is why we're using a temporary definition to set up `\ekvcFlagSetTrue` and `\ekvcFlagSetFalse`.

(End definition for `\ekvcFlagRaise`. This function is documented on page 10.)
As already explained, truthiness is defined as a flag’s height is odd, so we just branch accordingly here.

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

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\}}

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 \@secondoftwo\}

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\texttt{(#1)}\}
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.

\begin{verbatim}
def\ekvcFlagGetHeights#1{
def\ekvc@flag@get@heights#1\ekv@stop{}
def\ekvc@flag@get@heights#1{
def\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}}
}
def\ekvc@flag@get@heights@done\ekv@stop
\expandafter\ekvc@flag@get@heights@\ekvcFlagHeight\ekv@stop\ekv@stop#1#2

{#2[1]}
\end{verbatim}

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

### 2.2.3 Helper Macros

A test which can be reduced to an if-empty by gobbling everything up to the first space.

\begin{verbatim}
def\ekvc@ifspace#1{
def\ekvc@ifspace@#1 \ekv@ifempty@B
\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B@firstoftwo
}
def\ekvc@ifspace@#1 % keep this space
{\ekv@ifempty@A\ekv@ifempty@B
}
\end{verbatim}

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

Used to test whether a token list contains exactly two \TeX{} arguments.

\begin{verbatim}
def\ekvc@ifnottwoargs#1{
def\ekvc@ifempty@gtwo#1{\ekv@ifempty@B}
\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B@firstoftwo
}
def\ekvc@ifempty@gtwo#1#2{\ekv@ifempty@B}
\end{verbatim}

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

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.

\begin{verbatim}
def\ekvc@extract@mark#1{
\end{verbatim}

(End definition for `\ekvc@extract@mark` and `\ekvc@extract@mark0`)

41
2.2.4 Assertions

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

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

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

2.2.5 Messages

Boring unexpandable error messages.

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

(End definition for \ekvc@assert@twoargs.)
\protected\def\ekvc@err@unknown@key#1\% 
\{ \errmessage \{expkv-cs Error: Unknown key `\unexpanded{#1}' for macro `\ekvc@set'}\%
\} \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 `\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
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}}
(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.

Symbols
\# ........................................ 934
\\ ....................................... 935
A
aggregate ................................. 7
alias ..................................... 7
D
default ................................. 7
E
\ekvcChange ............................ 8, 555
\ekvDate ................................. 16, 6, 13, 21
\ekvcFlagGetHeight ........................ 11, 899
\ekvcFlagGetHeights ........................ 11, 902
\ekvcFlagHeight ........................ 749, 813, 843, 846, 848, 900, 909, 915
\ekvcFlagIf ............................. 11, 868
\ekvcFlagIfRaised ........................ 11, 875
\ekvcFlagRaise .......................... 10, 788, 802, 814
\ekvcFlagReset .......................... 11, 882
\ekvcFlagGetFalse ........................ 10, 808, 845
\ekvcFlagGetTrue ........................ 10, 806, 845
\ekvcHash ............................... 4337
\ekvcHashAndForward ........................ 5330
\ekvcHashAndUse .......................... 5301
\ekvcPass ............................... 14, 599
\ekvcSecondaryKeys ........................ 6, 532
\ekvSplit ............................... 392
\ekvSplitAndForward ........................ 4385
\ekvSplitAndUse .......................... 4355
\ekvValue ............................... 5507
\ekvValueFast ............................ 5514
\ekvValueSplit ........................... 5515
\ekvValueSplitFast ........................ 6526
\ekvcVersion ............................ 16, 6, 13, 21
\ekvdef ............................... 740, 782, 789
\ekvdefNoVal ............................ 803
\ekverr ............................... 983
\ekvifdefefined .......................... 601, 648, 664, 690
\ekvifdefefinedNoVal ........................ 654
\ekvifdefinedset .......................... 557
\ekvlet ............................... 184, 398, 623, 727, 769
\ekvletkv ............................... 650
\ekvletkvNoVal ........................... 656
\ekvletNoVal ............................. 629, 672
\ekvoptarg .............................. 12
\ekvoptargFF ............................. 12
\ekvparse .............................. 137, 355, 535
\ekvset ............................... 43, 582, 633
F
flag-bool ............................. 8
flag-false ............................. 8
flag-raise ............................. 8
flag-true ............................. 8
L
\lccode ............................. 934.935
long ..................................... 6
\lowercase ............................... 936
M
meta ................................. 7
N
nmeta ................................. 7
P
process ............................... 14
T
\TeX and \LaTeX commands:
\kva@SetupHashKeys ........................ 351
\ekv@alignsafe ............................ 51
\ekv@emptysafe ............................ 28
\kva@endalignsafe ........................ 53
\ekv@exparg .............................. 579, 803, 840, 875, 983
\ekv@expargtwice .......................... 821
\ekv@firstofone ........................... 861
\ekv@firstoftwos ........................... 874, 871, 878
\ekv@firstofthre ........................... 34, 124, 854
\ekv@gobbleto@stop .......................... 908
\ekv@ifdefefined .......................... 149, 281
\kva@ifempty .............................. 366, 439, 510, 517, 546, 549, 616, 791
\ekv@isempty ............................ 595, 716
\ekv@isempty@ ................................ 924, 931
\ekv@isempty@A ............................ 920, 924, 929, 931
\ekv@isempty@B ............................ 919, 920, 928, 929
\ekv@isempty@false ........................ 920, 929
\ekv@name 207, 209, 419, 421, 602, 670, 695
\ekv@stop 566, 569, 709, 710, 713, 715, 794, 828, 832, 836, 846, 848, 851, 858, 888, 891, 896, 900, 901, 904, 908, 909, 911, 912, 914, 915, 932, 937
\ekv@after@ptype \err unknown key \err no key macro \err invalid bool \err empty hash
\ekv@defarggobbler
\ekv@change@iflong
\ekv@change@ifnottwoargs
\ekv@assert@not@long
\ekv@assert@twoargs
\ekv@change
\ekv@change@a
\ekv@change@b
\ekv@change@c
\ekv@change
\ekv@assert@not@expanded
\ekv@assert@not@expanded@a
\ekv@assert@not@expanded@b
\ekv@err . 983, 984, 985, 986, 987, 989
\ekv@err@already@defined
\ekv@err@empty@hash
\ekv@err@invalid@bool
\ekv@err@missing@hash
\ekv@err@missing@type
\ekv@err@un@key@macro
\ekv@err@un@long
\ekv@err@un@two
\ekv@err@toomany
\ekv@err@unknown@hash
\ekv@err@unknown@key
\ekv@flag@get@height@single
\ekv@flag@get@heights
\ekv@flag@get@heights@0
\ekv@flag@get@heights@done
\ekv@flag@height
\ekv@flag@height@0
\ekv@flag@name
\ekv@flag@namescheme
\ekv@SetupHashKeys@b
\ekv@SetupHashKeys@a
\ekv@setup@splitmacro
\ekv@setup@hashmacro
\safesplithash
\safehash
\long keycount
\ifempty@gtwo
\ifnottwoargs
\ifspace
\ifnottwoargs
\ekv@defaultarg
\ekv@setup@splitmacro
\ekv@setup@hashmacro