exPkv|DEF

a key-defining frontend for exPkv

Jonathan P. Spratte∗

2021-05-24 v0.8a

Abstract

exPkv|DEF provides a small \langle key\rangle=\langle value\rangle interface to define keys for exPkv. Key-types are declared using prefixes, similar to static typed languages. The stylised name is exPkv|DEF but the files use expkv-def, 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 Macros ................................................................. 2
  1.2 Prefixes ................................................................. 2
    1.2.1 p-Prefixes ........................................................... 2
    1.2.2 t-Prefixes ............................................................ 3
  1.3 Bugs ................................................................. 7
  1.4 Example ............................................................ 8
  1.5 License ............................................................. 9

2 Implementation 10
  2.1 The \LaTeX{} Package ......................................................... 10
  2.2 The Generic Code ......................................................... 10
    2.2.1 Key Types ........................................................... 12
    2.2.2 Key Type Helpers .................................................... 24
    2.2.3 Handling also ....................................................... 25
    2.2.4 Tests .............................................................. 26
    2.2.5 Messages .......................................................... 29

Index 32

∗jspratte@yahoo.de
1 Documentation

Since the trend for the last couple of years goes to defining keys for a ⟨key⟩=⟨value⟩ interface using a ⟨key⟩=⟨value⟩ interface, I thought that maybe providing such an interface for expkv will make it more attractive for actual use, besides its unique selling points of being fully expandable, and fast and reliable. But at the same time I don’t want to widen expkv’s initial scope. So here it is expkv def, go define ⟨key⟩=⟨value⟩ interfaces with ⟨key⟩=⟨value⟩ interfaces.

Unlike many of the other established ⟨key⟩=⟨value⟩ interfaces to define keys, expkv def works using prefixes instead of suffixes (e.g., .tl_set:N of l3keys) or directory like handlers (e.g., ./store in of pgfkeys). This was decided as a personal preference, more over in TEX parsing the first space is way easier than parsing for the last one. expkv def’s prefixes are sorted into two categories: p-type, which are equivalent to TEX’s prefixes like \long, and t-type defining the type of the key. For a description of the available p-prefixes take a look at subsubsection 1.2.1, the t-prefixes are described in subsubsection 1.2.2.

expkv def 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

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

1.1 Macros

Apart from version and date containers there is only a single user-facing macro, and that should be used to define keys.

\begin{verbatim}
\ekvdefinekeys \ekvdefinekeys{(set)}{(key)=(value), ...}
\end{verbatim}

In ⟨set⟩, 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.

\begin{verbatim}
\ekvdDate \ekvdVersion
\end{verbatim}

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

1.2 Prefixes

As already said there are p-prefixes and t-prefixes. Not every p-prefix is allowed for all t-prefixes.

1.2.1 p-Prefixes

The two p-type prefixes \long and \protected are pretty simple by nature, so their description is pretty simple. They affect the ⟨key⟩ at use-time, so omitting \long doesn’t mean that a ⟨definition⟩ can’t contain a \par token, only that the ⟨key⟩ will not accept
a `\par` in `(value)`. On the other hand `new` and `also` might be simple on first sight as well, but their rules are a bit more complicated.

**also**

The following key type will be *added* to an existing `(key)`'s definition. You can’t add a type taking an argument at use time to an existing key which doesn’t take an argument and vice versa. Also you’ll get an error if you try to add an action which isn’t allowed to be either `long` or `protected` to a key which already is `long` or `protected` (the opposite order would be suboptimal as well, but can’t be really captured with the current code).

A key already defined as `long` or `protected` will stay `long` or `protected`, but you can as well add `long` or `protected` with the `also` definition.

As a small example, suppose you want to create a boolean key, but additionally to setting a boolean value you want to execute some more code as well, you can use the following

```latex
\ekvdefinekeys{also−example}
{ bool key = \ifmybool, also code key = \domystuff[#1] }
```

If you use `also` on a choice, `bool`, `invbool`, or `boolpair` key it is tried to determine if the key already is of one of those types. If this test is true the declared choices will be added to the possible choices but the key’s definition will not be changed other than that. If that wouldn’t have been done, the callbacks of the different choices could get called multiple times.

**protected**

The following key will be defined `\protected`. Note that key-types which can’t be defined expandable will always use `\protected`.

**long**

The following key will be defined `\long`.

**new**

The following key must be new (so previously undefined). An error is thrown if it is already defined and the new definition is ignored. `new` only asserts that there are no conflicts between `NoVal` keys and other `NoVal` keys or value taking keys and other value taking keys. For example you can use the following without an error:

```latex
\ekvdefinekeys{new−example}
{ code key = \domystuffwitharg[#1], new noval key = \domystuffwithoutarg }
```

### 1.2.2 t-Prefixes

Since the p-type prefixes apply to some of the t-prefixes automatically but sometimes one might be disallowed we need some way to highlight this behaviour. In the following
an enforced prefix will be printed black (protected), allowed prefixes will be grey (protected), and disallowed prefixes will be red (protected). This will be put flush-right in the syntax showing line.

```latex
\begin{verbatim}
code (key) = { (definition) }  \textit{new also protected long}
\end{verbatim}
```

Define \textit{(key)} to expand to \textit{(definition)}. The \textit{(key)} will require a \textit{(value)} for which you can use \texttt{#1} inside \textit{(definition)}. The code variant will fully expand \textit{(definition)} inside an \texttt{edef}.

```latex
\begin{verbatim}
noval (key) = { (definition) }  \textit{new also protected long}
\end{verbatim}
```

The noval type defines \textit{(key)} to expand to \textit{(definition)}. The \textit{(key)} will not take a \textit{(value)}. enoval fully expands \textit{(definition)} inside an \texttt{edef}.

```latex
\begin{verbatim}
default (key) = { (definition) }  \textit{new also protected long}
\end{verbatim}
```

This serves to place a default \textit{(value)} for a \textit{(key)} that takes an argument, the \textit{(key)} can be of any argument-grabbing kind, and when used without a \textit{(value)} it will be passed \textit{(definition)} instead. The qdefault variant will expand the \textit{(key)}'s code once, so will be slightly quicker, but not change if you redefine \textit{(key)}. odefault is just another name for qdefault. The fdefault version will expand the key code until a non-expandable token or a space is found, a space would be gobbled.\footnote{For those familiar with \TeX-coding: This uses a \texttt{\romannumeral} expansion.} The edefault on the other hand fully expands the \textit{(key)}-code with \textit{(definition)} as its argument inside of an \texttt{edef}.

```latex
\begin{verbatim}
initial (key) = { (value) }  \textit{new also protected long}
\end{verbatim}
```

With initial you can set an initial \textit{(value)} for an already defined argument taking \textit{(key)}. It'll just call the key-macro of \textit{(key)} and pass it \textit{(value)}. The einitial variant will expand \textit{(value)} using an \texttt{edef} expansion prior to passing it to the key-macro and the oinitial variant will expand the first token in \textit{(value)} once. finitial will expand \textit{(value)} until a non-expandable token or a space is found, a space would be gobbled.\footnote{Again using \texttt{\romannumeral} coding.}

```latex
\begin{verbatim}
bool (key) = (cs)
\end{verbatim}
```

The \textit{(cs)} should be a single control sequence, such as \texttt{\iffoo}. This will define \textit{(key)} to be a boolean key, which only takes the values \texttt{true} or \texttt{false} and will throw an error for other values. If the key is used without a \textit{(value)} it'll have the same effect as if you use \texttt{(key)=true}. bool and gbool will behave like \TeX-ifs so either \texttt{\iftrue} or \texttt{\iffalse}. The boolTF and gboolTF variants will both take two arguments and if true the first will be used else the second, so they are always either \texttt{\iftrue} or \texttt{\iffalse}. The variants with a leading \texttt{g} will set the control sequence globally, the others locally. If \texttt{(cs)} is not yet defined it’ll be initialised as the \texttt{false} version. Note that the initialisation is not done with \texttt{\newif}, so you will not be able to do \texttt{\iftrue} outside of the \texttt{(key)=(value)} interface, but you could use \texttt{\newif} yourself. Even if the \texttt{(key)} will not be \texttt{\protected} the commands which execute the \texttt{true} or \texttt{false} choice will be, so the usage should be safe in an expansion context (e.g., you can use edefault \textit{(key)} = \texttt{false} without an issue to change the default behaviour to execute the \texttt{false} choice). Internally a bool \textit{(key)} is the same as a choice key which is set up to handle \texttt{true} and \texttt{false} as choices.
invbool
\texttt{invbool}
\texttt{invboolTF}

\texttt{gboolpairTF}
\texttt{gbboolpairTF}

boolpair
\texttt{boolpair}
\texttt{boolpairTF}

These are inverse boolean keys, they behave like bool and friends but set the opposite meaning to the macro \texttt{cs} in each case. So if \texttt{key=true} is used \texttt{invbool} will set \texttt{cs} to false and vice versa.

boolpair \langle key \rangle = \langle cs \rangle
\texttt{new also protected long}

The boolpair key type behaves like both bool and invbool, the \texttt{cs\textsubscript{1}} will be set to the meaning according to the rules of bool, and \texttt{cs\textsubscript{2}} will be set to the opposite.

store \langle key \rangle = \langle cs \rangle
\texttt{new also protected long}

The \texttt{cs} should be a single control sequence, such as \texttt{foo}. This will define \texttt{key} to store \texttt{value} inside of the control sequence. If \texttt{cs} isn't yet defined it will be initialised as empty. The variants behave similarly to their \texttt{def,edef,gdef, and xdef} counterparts, but store and gstore will allow you to store macro parameters inside of them by using \texttt{unexpanded}.

data \langle key \rangle = \langle cs \rangle
\texttt{new also protected long}

The \texttt{cs} should be a single control sequence, such as \texttt{foo}. This will define \texttt{key} to store \texttt{value} inside of the control sequence. But unlike the store type, the macro \texttt{cs} will be a switch at the same time, it'll take two arguments and if \texttt{key} was used expands to the first argument followed by \texttt{value} in braces, if \texttt{key} was not used \texttt{cs} will expand to the second argument (so behave like \texttt{\@secondoftwo}). The idea is that with this type you can define a key which should be typeset formatted. The edata and xdata variants will fully expand \texttt{value}, the gdatalong global. The p- prefixes will only affect the key-macro, \texttt{cs} will always be expandable and \texttt{long}.

dataT \langle key \rangle = \langle cs \rangle
\texttt{new also protected long}

Just like data, but instead of \texttt{cs} grabbing two arguments it'll only grab one, so by default it'll behave like \texttt{\@gobble}, and if a \texttt{value} was given to \texttt{key} the \texttt{cs} will behave like \texttt{\@firstofone} appended by \{\texttt{value}\}.

int \langle key \rangle = \langle cs \rangle
\texttt{new also protected long}

The \texttt{cs} should be a single control sequence, such as \texttt{foo}. An int key will be a \LaTeX{}\-count register. If \texttt{cs} isn't defined yet, \texttt{\newcount} will be used to initialise it. The \texttt{eint} and \texttt{xint} versions will use \texttt{\numexpr} to allow basic computations in their \texttt{value}. The \texttt{gint} and \texttt{xint} variants set the register globally.

dimen \langle key \rangle = \langle cs \rangle
\texttt{new also protected long}

The \texttt{cs} should be a single control sequence, such as \texttt{foo}. This is just like \texttt{int} but uses a dimen register, \texttt{\newdimen} and \texttt{\dimexpr} instead.

skip \langle key \rangle = \langle cs \rangle
\texttt{new also protected long}

The \texttt{cs} should be a single control sequence, such as \texttt{foo}. This is just like \texttt{int} but uses a skip register, \texttt{\newskip} and \texttt{\glueexpr} instead.
toks \langle key \rangle = \{ cs \}

doesn’t def
gapptoks
apptoks
gapptoks

toks \langle key \rangle = \{ cs \}

The \langle cs \rangle should be a single control sequence, such as \texttt{\textbackslash foo}. Store \langle value \rangle inside of a toks-register. The g variants use \texttt{\global}, the app variants append \langle value \rangle to the contents of that register. If \langle cs \rangle is not yet defined it will be initialised with \texttt{\newtoks}.

box box \langle key \rangle = \{ cs \}

The \langle cs \rangle should be a single control sequence, such as \texttt{\textbackslash foo}. Typesets \langle value \rangle into a \texttt{\hbox} and stores the result in a box register. The boxes are colour safe. \texttt{\expandafter} doesn’t provide a vbox type.

meta meta \langle key \rangle = \{ \{ key \} = \{ value \}, \ldots \}

This key type can set other keys, you can access the \langle value \rangle which was passed to \langle key \rangle inside the \langle key \rangle = \{ value \} list with \#1. It works by calling a sub-\texttt{\ekvset} on the \langle key \rangle = \{ value \} list, so a set key will only affect that \langle key \rangle = \{ value \} list and not the current \texttt{\ekvset}. Since it runs in a separate \texttt{\ekvset} you can’t use \texttt{\ekvset} aka using keys or similar macros in the way you normally could.

nmeta nmeta \langle key \rangle = \{ \langle key \rangle = \{ value \}, \ldots \}

This key type can set other keys, the difference to meta is, that this key doesn’t take a value, so the \langle key \rangle = \{ value \} list is static.

smeta smeta \langle set \rangle = \{ \langle key \rangle = \{ value \}, \ldots \}

Yet another meta variant. An smeta key will take a \langle value \rangle which you can access using \#1, but it sets the \langle key \rangle = \{ value \} list inside of \langle set \rangle, so is equal to \texttt{\ekvset}(\langle set \rangle)\{ \langle key \rangle = \{ value \}, \ldots \}.

snmeta snmeta \langle set \rangle = \{ \langle set \rangle \} = \{ \langle key \rangle = \{ value \}, \ldots \}

And the last meta variant. snmeta is a combination of smeta and nmeta. It doesn’t take an argument and sets the \langle key \rangle = \{ value \} list inside of \langle set \rangle.

set set \langle key \rangle = \{ \langle set \rangle \}

This will define \langle key \rangle to change the set of the current \texttt{\ekvset} invocation to \langle set \rangle. You can omit \langle set \rangle (including the equals sign), which is the same as using \texttt{\set} \langle key \rangle = \{ \{ key \} \}. The created set key will not take a \langle value \rangle. Note that just like in expandable \texttt{\expandafter} it’ll not be checked whether \langle set \rangle is defined and you’ll get a low-level \texttt{\TeX} error if you use an undefined \langle set \rangle.

choice choice \langle key \rangle = \{ \langle value \rangle = \{ definition \}, \ldots \}

Defines \langle key \rangle to be a choice key, meaning it will only accept a limited set of values. You should define each possible \langle value \rangle inside of the \langle value \rangle = \{ definition \} list. If a defined \langle value \rangle is passed to \langle key \rangle the \langle definition \} will be left in the input stream. You can make individual values protected inside the \langle value \rangle = \{ definition \} list. By default a choice key is expandable, an undefined \langle value \rangle will throw an error in an expandable way (but see the unknown-choice prefix). You can add additional choices after the \langle key \rangle was created by using choice again for the same \langle key \rangle, redefining choices is possible the same way, but there is no interface to remove certain choices.
unknown-choice

unknown-choice (key) = {{definition}}

By default an unknown value passed to a choice or bool key will throw an error. However, with this prefix you can define an alternative action which should be executed if (key) received an unknown choice. In (definition) you can refer to the choice which was passed in with #1.

unknown_code

unknown code = {{definition}}

By default expkv throws errors when it encounters unknown keys in a set. With the unknown prefix you can define handlers that deal with undefined keys, instead of a (key) name you have to specify a subtype for this prefix, here the subtype is code.

With unknown code the (definition) is used for unknown keys which were provided a value (so corresponds to \ekvdefunknown), you can access the key name with #1 and the value with #2.\footnote{There is some trickery involved to get this more intuitive argument order without any performance hit if you compare this to \ekvdefunknown directly.}

unknown_noval

unknown noval = {{definition}}

This is like unknown code but uses (definition) for unknown keys to which no value was passed (so corresponds to \ekvdefunknownNoVal). You can access the key name with #1.

unknown_redirect-code

unknown redirect-code = {{set-list}}

This uses a predefined action for unknown code. Instead of throwing an error, it is tried to find the (key) in each (set) in the comma separated (set-list). The first found match will be used and the remaining options from the list discarded. If the (key) isn't found in any (set) an expandable error will be thrown eventually. Internally expkv's \ekvredirectunknown will be used.

unknown_redirect-noval

unknown redirect-noval = {{set-list}}

This behaves just like unknown redirect-code but will set up means to forward keys for unknown noval. Internally expkv's \ekvredirectunknownNoVal will be used.

unknown_redirect

unknown redirect = {{set-list}}

This is a short cut to apply both, unknown redirect-code and unknown redirect-noval, as a result you might get doubled error messages, one from each.

1.3 Bugs

I don't think there are any (but every developer says that), if you find some please let me know, either via the email address on the first page or on GitHub: https://github.com/Skillmon/tex_expkv-def
1.4 Example

The following is an example code defining each base key-type once. Please admire the very creative key-name examples.

\ekvdefinekeys \{example\}
{
  \longcode keyA = \#1
  \nval keyA = NoVal given
  \bool keyB = \keyB
  \boolTF keyC = \keyC
  \store keyD = \keyD
  \data keyE = \keyE
  \dataT keyF = \keyF
  \int keyG = \keyG
  \dimen keyH = \keyH
  \skip keyI = \keyI
  \toks keyJ = \keyJ
  \default keyJ = \emptytest
  \newbox keyK = \keyK
  \qdefault keyK = K
  \choice keyL =
    \\{\\protected\ 1 = \texttt{\{a}}
    ,2 = b
    ,3 = c
    ,4 = d
    ,5 = e
  \}
  \edefault keyL = 2
  \meta keyM = \{keyA=\#1, keyB=false\}
  \invbool keyN = \keyN
  \boolpair keyO = \keyOa\keyOb
}

Since the data type might be a bit strange, here is another usage example for it.

\ekvdefinekeys \{ex\}
{
  \data name = \Pname
  \data age = \Page
  \dataT hobby = \Phobby
}
\newcommand\Person[1]{
  \begin{description}
    \item[\Pname] {\errmessage[\text{A person requires a name}]} \n    \item[Age] \Page{\textit}{\errmessage[\text{A person requires an age}]} \n    \item[Hobbies] \n  \end{description}
}

8
In this example a person should have a name and an age, but doesn’t have to have hobbies. The name will be displayed as the description item and the age in Italics. If a person has no hobbies the description item will be silently left out. The result of the above code looks like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Hobbies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jonathan P. Spratte</td>
<td>young</td>
<td>\TeX\ coding</td>
</tr>
<tr>
<td>Some User</td>
<td>unknown</td>
<td>Reading Documentation</td>
</tr>
<tr>
<td>Anybody</td>
<td>any</td>
<td></td>
</tr>
</tbody>
</table>

1.5 License

Copyright © 2020–2021 Jonathan P. Spratte

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

This work is “maintained” (as per LPPL maintenance status) by Jonathan P. Spratte.
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}
\input{expkv-def.tex}
\end{verbatim}

2.2 The Generic Code

The rest of this implementation will be the generic code.

Load \texttt{expkv} if the package didn’t already do so – since \texttt{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 \texttt{expkv} to save us from retyping them.

\begin{verbatim}
\input expkv
\end{verbatim}

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

\begin{verbatim}
\expandafter\ifx\csname ekvdVersion\endcsname\relax
\else
\expandafter\endinput
\fi
\end{verbatim}

\begin{verbatim}
\def\ekvdVersion{0.8a}
\def\ekvdDate{2021-05-24}
\end{verbatim}

\texttt{\ekvdVersion} \texttt{\ekvdDate} We’re on our first input, so lets store the version and date in a macro.

\begin{verbatim}
\def\ekvdVersion{0.8a}
\def\ekvdDate{2021-05-24}
\end{verbatim}

(End definition for \texttt{\ekvdVersion} and \texttt{\ekvdDate}. These functions are documented on page 2.)

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

\begin{verbatim}
\ecsname ekvd@tmp\endcsname
\end{verbatim}

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

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

\texttt{\ekvd@tmp} will be reused later to handle expansion during the key defining. But we don’t need it to ever store information long-term after \texttt{expkv-def} was initialized.
\protected\def\ekvd@clear@prefixes
{\
  \let\ekvd@long\ekvd@empty\
  \let\ekvd@prot\ekvd@empty\
  \let\ekvd@ifalso\@secondoftwo\
  \long\def\ekvd@ifnew##1##2##3{##3}\
}\ekvd@clear@prefixes

(End definition for \ekvd@long and others.)

\ekvdefinekeys  \ekvdefinekeys This is the one front-facing macro which provides the interface to define keys. It's using \ekvparse to handle the \texttt{(key)=(value)} list, the interpretation will be done by \ekvd@noarg and \ekvd@arg. The \texttt{(set)} for which the keys should be defined is stored in \ekvd@set.
\protected\def\ekvdefinekeys#1%\%{\
  \def\ekvd@set{#1}\
  \ekvparse\ekvd@noarg\ekvd@arg
}\ekvdefinekeys

(End definition for \ekvdefinekeys. This function is documented on page 2.)

\ekvd@noarg  \ekvd@noarg and \ekvd@arg store whether there was a value in the \texttt{(key)=(value)} pair. \ekvd@handle has to test whether there is a space inside the key and if so calls the prefix grabbing routine, else we throw an error and ignore the key.
\protected\def\ekvd@noarg#1%\%{\
  \let\ekvd@ifnoarg\@firstoftwo\
  \ekvd@handle{#1}{}\%
}\ekvd@noarg

(End definition for \ekvd@noarg, \ekvd@arg, and \ekvd@handle.)

\ekvd@prefix  \ekvd@prefix\@s@er separates prefixes into two groups, the first being prefixes in the \TeX{} sense (\texttt{long} and \texttt{protected}) which use \texttt{@p@} in their name, the other being key-types (\texttt{code}, \texttt{int}, \texttt{etc.}) which use \texttt{@t@} instead. \ekvd@prefix\@s@er splits at the first space and checks whether its a \texttt{@p@} or \texttt{@t@} type prefix. If it is neither throw an error and gobble the definition (the value).
\protected\def\ekvd@prefix\#1 {\ekv@\strip\ekvd@prefix\#1\ekv@mark}

(End definition for \ekvd@noarg, \ekvd@arg, and \ekvd@handle.)
\ekv@prefix@after@p

The \texttt{\@p@} type prefixes are all just modifying a following \texttt{\@t@} type, so they will need to search for another prefix. This is true for all of them, so we use a macro to handle this. It'll throw an error if there is no other prefix.

\protected\def\ekvd@prefix@after@p#1{\ekvd@ifspace{#1}{\ekvd@prefix#1\ekv@stop}{\ekvd@err@missing@type\@gobble}}

\ekvd@p@long\ekvd@p@protected\ekvd@p@protect\ekvd@p@also\ekvd@p@new

Define the \texttt{\@p@} type prefixes, they all just store some information in a temporary macro.

\protected\def\ekvd@p@long{\let\ekvd@long\long}\protected\def\ekvd@p@protected{\let\ekvd@prot\protected}\let\ekvd@p@protect\ekvd@p@protected\protected\def\ekvd@p@also{\let\ekvd@ifalso\@firstoftwo}\protected\def\ekvd@p@new{\let\ekvd@ifnew\ekvd@assert@new}

\ekvd@type@set\ekvd@t@set

The \texttt{\@set@} type is quite straightforward, just define a \texttt{NoVal} key to call \texttt{\ekvchangeset}.

\protected\def\ekvd@type@set#1#2{\ekvd@assert@not@long\ekvd@assert@not@protected\ekvd@ifnew{NoVal}{#1}{\ekv@expargtwice{\ekvd@add@noval{#1}}{\ekvchangeset{#2}}\ekvd@assert@not@protected@also}{\ekv@expargtwice{\ekvdefNoVal\ekvd@set{#1}}{\ekvchangeset{#2}}}
Another pretty simple type, noval just needs to assert that there is a definition and that long wasn't specified. There are types where the difference in the variants is so small, that we define a common handler for them, those common handlers are named with \@type@. noval and enoval are so similar that we can use such a \@type@ macro, even if we could've done noval in a slightly faster way without it.

\protected\long\def\ekvd@type@noval#1#2#3{% 
\ekvd@ifnew{NoVal}{#2}{% 
\ekvd@assert@arg{% 
\ekvd@assert@not@long 
\ekvd@prot\ekvd@long#1\ekvd@tmp##1{#3}% 
\ekvd@ifalso{% 
\ekv@exparg{\ekvd@add@noval{#2}}{\ekvd@tmp{##1}}% 
\ekvletNoVal\ekvd@set{#2}\ekvd@tmp}% 
}% 
}\ekvd@assert@arg{\% 
\ekvd@prot\ekvd@long#1\ekvd@tmp#1{%3}% 
\ekvd@ifalso{% 
\ekv@exparg{\ekvd@add@noval{#2}}{\ekvd@tmp{#1}}% 
\ekvlet\ekvd@set{#2}\ekvd@tmp}% 
}\% 
}% 
}\protected\def\ekvd@t@noval{\ekvd@type@noval\def} 
\protected\def\ekvd@t@enoval{\ekvd@type@noval\edef} 

(End definition for \ekvd@type@noval, \ekvd@t@noval, and \ekvd@t@enoval.)

code is simple as well, ecode has to use \edef on a temporary macro, since expr doesn't provide an \ekvedef.

\protected\long\def\ekvd@type@code#1#2#3{% 
\ekvd@ifnew{\% 
\ekvd@assert@arg{% 
\ekvd@prot\ekvd@long#1\ekvd@tmp#1{%3}% 
\ekvd@ifalso{% 
\ekv@exparg{\ekvd@add@val{#2}}{\ekvd@tmp{#1}}% 
\ekvlet\ekvd@set{#2}\ekvd@tmp}% 
}\% 
}% 
}\protected\def\ekvd@t@code{\ekvd@type@code\def} 
\protected\def\ekvd@t@ecode{\ekvd@type@code\edef} 

(End definition for \ekvd@type@code, \ekvd@t@code, and \ekvd@t@ecode.)
\ekvd@type@default asserts there was an argument, also the key for which one wants to set a default has to be already defined (this is not so important for default, but qdefault requires is). If everything is good, \edef a temporary macro that expands \ekvd@set and the \csname for the key, and in the case of qdefault does the first expansion step of the key-macro.

\begin{verbatim}
\protected\long\def\ekvd@type@default#1#2#3#4\
  {\
    \ekvd@assert@arg\
    \ekvfdefined\ekvd@set{#3}\
    \ekvd@assert@not@new\
    \ekvd@assert@not@long\
    \prot\edef\ekvd@tmp\
      \\unexpanded\expandafter#1\
        {#2\csname\ekv@name\ekvd@set{#3}\endcsname{#4}}\%
    \ekvd@ifalso\{\ekv@exparg{\ekvd@add@noval{#3}}\ekvd@tmp{}\%
    \ekvletNoVal\ekvd@set{#3}\ekvd@tmp\%
    \}\%
  }
\protected\def\ekvd@t@default{\ekvd@type@default{}{}}
\protected\def\ekvd@t@qdefault{\ekvd@type@default{\expandafter\expandafter}}
\let\ekvd@t@odefault\ekvd@t@qdefault
\protected\def\ekvd@t@fdefault{\ekvd@type@default{}{\romannumeral'\^^@}}
\end{verbatim}

(End definition for \ekvd@type@default and others.)

\ekvd@edefault edefault is too different from default and qdefault to reuse the @type@ macro, as it doesn't need \unexpanded macro inside of \edef.

\begin{verbatim}
\protected\long\def\ekvd@t@edefault#1#2\
  {\
    \ekvfdefined\ekvd@set{#1}\
    \ekvd@assert@arg\
    \ekvfdefined\ekvd@set{#1}\
    \ekvd@assert@not@new\
    \ekvd@assert@not@long\
    \prot\edef\ekvd@tmp\
      \\unexpanded\expandafter#1\
        {#2\csname\ekv@name\ekvd@set{#1}\endcsname{#2}}\%
    \ekvd@ifalso\{\ekv@exparg{\ekvd@add@noval{#1}}\ekvd@tmp{}\%
    \ekvletNoVal\ekvd@set{#1}\ekvd@tmp\%
    \}\%
  }
\end{verbatim}

(End definition for \ekvd@t@edefault.)
The boolean types are a quicker version of a choice that accept true and false, and set up the NoVal action to be identical to \(\langle key\rangle=true\). The true and false actions are always just \let\ting the macro in #7 to some other macro (e.g., \iftrue).

(End definition for \ekvd@t@initial and others.)
\protected\def\ekvd@t@invbool{\ekvd@type@bool{}\iffalse\iftrue}
\protected\def\ekvd@t@ginvbool{\ekvd@type@bool\global\iffalse\iftrue}
\protected\def\ekvd@t@invboolTF{\ekvd@type@bool{}\@secondoftwo\@firstoftwo}
\protected\def\ekvd@t@ginvboolTF{\ekvd@type@bool\global\@secondoftwo\@firstoftwo}

(End definition for \ekvd@type@bool and others.)

\ekvd@type@boolpair
\ekvd@t@boolpair
\ekvd@t@gboolpair
\ekvd@t@boolpairTF
\ekvd@t@gboolpairTF

The boolean pair types are essentially the same as the boolean types, but set two macros instead of one.

\protected\def\ekvd@type@boolpair#1#2#3#4#5#6{% \ekvd@ifnew{}{#4}{% \ekvd@newlet#5#3% \ekvd@newlet#6#2% \ekvd@type@choice{#4}% \protected\ekvdefNoVal\ekvd@set{#4}{#1\let#5#2#1\let#6#3}% \protected\expandafter\def\csname\ekvd@choice@name\ekvd@set{#4}{true}\endcsname{#1\let#5#2#1\let#6#3} \protected\expandafter\def\csname\ekvd@choice@name\ekvd@set{#4}{false}\endcsname{#1\let#5#3#1\let#6#2} % }\}
\protected\def\ekvd@t@boolpair#1#2{\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair{}\iftrue\iffalse{#1}#2}}
\protected\def\ekvd@t@gboolpair#1#2{\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair\global\iftrue\iffalse{#1}#2}}
\protected\def\ekvd@t@boolpairTF#1#2{\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair{}\@firstoftwo\@secondoftwo{#1}#2}}
\protected\def\ekvd@t@gboolpairTF#1#2{\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair\global\@firstoftwo\@secondoftwo{#1}#2}}

(End definition for \ekvd@type@boolpair and others.)

\ekvd@type@data
\ekvd@t@data
\ekvd@t@dataT
\ekvd@t@gdata
\ekvd@t@gdataT

\protected\def\ekvd@type@data#1#2#3#4#5#6{% \ekvd@ifnew{}{#5}{% \ekvd@assert@filledarg{#6}{% \ekvd@newlet#6#1%}
\ekvd@ifalso
  {%
    \let\ekvd@prot\protected
    \ekvd@add@val(\#5){\long\#6\#1\#3\#4}\%
  \}
\%
\protected\ekvd@long\ekvdef\ekvd@set(\#5)%
  {\long\#6\#1\#3\#4}
\%

\ekvd@type@box Set up our boxes. Though we're a generic package we want to be colour safe, so we put an additional grouping level inside the box contents, for the case that someone uses color. \ekvd@newreg is a small wrapper which tests whether the first argument is defined and if not does \csname new#2\endcsname#1.

\protected\def\ekvd@type@box#1#2#3%
  {%
    \ekvd@ifnew{}{#2}%
    {%\ekvd@assert@filledarg{#3}%
      \ekvd@newreg{\box}#3%
      \ekvd@ifalso
        {%\let\ekvd@prot\protected
          \ekvd@add@val{\#2}{\#1\setbox\#3\hbox{\begingroup##1\endgroup}}%\}
        {%\protected\ekvd@long\ekvdef\ekvd@set{\#2}%
          {\#1\setbox\#3\hbox{\begingroup\#1\endgroup}}}%
      }
    }
  }
\protected\def\ekvd@t@box{\ekvd@type@box{}}
\protected\def\ekvd@t@gbox{\ekvd@type@box\global}

(End definition for \ekvd@type@data and others.)

\ekvd@type@toks Similar to box, but set the toks.

\ekvd@type@toks
\ekvd@t@toks
\ekvd@t@gtoks
\protected\def\ekvd@type@toks#1#2#3\%%
\ekvd@ifnew{}{#2}\%
\ekvd@assert@filledarg[#3]\%
\ekvd@newreg#3{toks}\%
\ekvd@ifalso
\let\ekvd@prot\protected
\ekvd@add@val{#2}{#1#3{##1}}{}%
\}
\}
\protected\def\ekvd@t@toks{\ekvd@type@toks}{}
\protected\def\ekvd@t@gtoks{\ekvd@type@toks\global}
(End definition for \ekvd@type@toks, \ekvd@t@toks, and \ekvd@t@gtoks.)

\ekvd@type@apptoks
\ekvd@type@apptoks
\ekvd@type@apptoks

Just like \texttt{toks}, but expand the current contents of the \texttt{toks} register to append the new contents.

\protected\def\ekvd@type@apptoks#1#2#3\%%
\ekvd@ifnew{}{#2}\%
\ekvd@assert@filledarg[#3]\%
\ekvd@newreg#3{toks}\%
\ekvd@ifalso
\let\ekvd@prot\protected
\ekvd@add@val{#2}{#1#3\expandafter{\the#3##1}}{}%
\}
\}
\protected\def\ekvd@t@apptoks{\ekvd@type@apptoks}{}
\protected\def\ekvd@t@gapptoks{\ekvd@type@apptoks\global}
(End definition for \ekvd@type@apptoks, \ekvd@t@apptoks, and \ekvd@t@gapptoks.)

\ekvd@type@reg
\ekvd@type@int
\ekvd@type@oint
\ekvd@type@gint
\ekvd@type@xint
\ekvd@type@dimen
\ekvd@type@edimen
\ekvd@type@gdimen
\ekvd@type@xdimen
\ekvd@type@skip
\ekvd@type@eskip
\ekvd@type@gskip
\ekvd@type@xskip

The \texttt{\ekvd@type@reg} can handle all the types for which the assignment will just be \langle\texttt{register}\rangle=\langle\texttt{value}\rangle.
\ekvd@type@store\ekvd@t@store\ekvd@t@gstore

The none-expanding store types use an \edef or \xdef and \unexpanded to be able to also store \# easily.

\protected\def\ekvd@type@store#1#2#3#4%  
  \ekvd@ifnew{}{#3}%  
  \ekvd@assert@filledarg[#4]{}  
  \ekvd@newlet[#4]{\ekvd@empty}  
  \ekvd@ifalso  
  \let\ekvd@prot\protected  
  \ekvd@add@val{#3}{#1#4{#2}}{}  
  \protected\ekvd@long\ekvdef\ekvd@set{#3}{#1#4{#2}}\relax%  
}  
\protected\def\ekvd@t@store{\ekvd@type@store\edef{\unexpanded{##1}}}  
\protected\def\ekvd@t@gstore{\ekvd@type@store\xdef{\unexpanded{##1}}}  
\protected\def\ekvd@t@estore{\ekvd@type@store\edef{##1}}  
\protected\def\ekvd@t@xstore{\ekvd@type@store\xdef{##1}}

(End definition for \ekvd@type@store and others.)

\ekvd@type@meta\ekvd@type@meta@a\ekvd@type@meta@c
\ekvd@t@meta\ekvd@t@nmeta

meta sets up things such that another instance of \ekvset will be run on the argument, with the same \langle set \rangle.

\protected\long\def\ekvd@type@meta#1#2#3#4#5#6#7%
\% \ekvd@ifnew{#1}#{6}\%
\% \ekvd@assert@filledarg#{7}\%
\% \edef\ekvd@tmp{\ekvd@set}\%
\expandafter{\ekvd@type@meta@a}\expandafter{\ekvd@tmp}#{7}\{%2\%
\ekvd@ifalso
{\ekv@exparg{#3}#{6}\{\ekvd@tmp}#{4}\%
{\csname ekvlet#1\endcsname}\ekvd@set}#{6}\ekvd@tmp\%
\%}
\%}
\protected\long\def{\ekvd@type@meta@b}\{\ekvset{#1}#2\%
\expandafter{\ekvd@type@meta@b}\expandafter{\ekvset}#{1}\%
\protected\def{\ekvd@type@meta@b}
\%\expandafter{\ekvd@type@meta@c}\expandafter
\% \ekvd@prot{\ekvd@long}\def{\ekvd@tmp}#{2}#1\%
\protected\def{\ekvd@type@meta}#{1}\{#1\}\ekvd@add@val{#1}#2\%
\% \ekvd@assert@not@long
\ekvd@type@meta{NoVal}\{\ekvd@add@noval}\ekvd@assert@not@long\also
\%}
\end{definition}

\ekvd@type@smeta
\ekvd@type@smeta@
\ekvd@t@smeta
\ekvd@t@snmeta
\ekvd@type@smeta
\ekvd@type@smeta@
\ekvd@t@smeta
\ekvd@t@snmeta

\ekvd@type@smeta
\ekvd@type@smeta@
\ekvd@t@smeta
\ekvd@t@snmeta

smeta is pretty similar to meta, but needs two arguments inside of ⟨value⟩, such that the first is the (set) for which the sub-⟨kvset⟩ and the second is the (key)=⟨value⟩ list.

\protected\long\def{\ekvd@type@smeta}#{1}#{2}#{3}#{4}#{5}#{6}#{7}%
\% \ekvd@ifnew{#1}#{6}\%
\% \ekvd@assert@twoargs#{7}\%
\% \edef\ekvd@tmp{\ekvd@meta@a}\%2\%
\ekvd@ifalso
{\ekv@exparg{#3}#{6}\{\ekvd@tmp}#{4}\%
{\csname ekvlet#1\endcsname}\ekvd@set}#{6}\ekvd@tmp\%
\%}
\%}
\protected\def{\ekvd@type@smeta}#{1}\{#1\}\ekvd@add@val{#1}#2\%
\protected\def{\ekvd@type@smeta}
\% \ekvd@assert@not@long
\ekvd@type@meta{NoVal}\{\ekvd@add@noval}\ekvd@assert@not@long\also
\%}

(End definition for \ekvd@type@meta and others.)
The choice type is by far the most complex type, as we have to run a sub-parser on the choice-definition list, which should support the \@type@ choice prefixes as well (but long will always throw an error, as they are not allowed to be long). \ekvd@type@choice will just define the choice-key, the handling of the choices definition will be done by \ekvd@populate@choice.

444 \protected\def\ekvd@type@choice#1\%
445 {%
446 \ekvd@assert@not@long
447 \ekvd@prot\edef\ekvd@tmp##1\%
448 {\unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#1}{##1}}}%
449 \ekvd@ifalso
450 {%
451 \ekvd@assert@val{#1}%
452 {%
453 \ekvd@if@not@already@choice{#1}%
454 {%
455 \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}\endcsname{##1}{#1}%
456 {\ekvd@tmp{##1}}%
457 {\ekvd@long\ekvdef}\ekvd@assert@not@long@also
458 }
459 }
460 }
461 {\ekvlet\ekvd@set{#1}\ekvd@tmp}
462%
463 }
464 }
465 {\ekvlet\ekvd@set{#1}\ekvd@tmp}%
466 }

\ekvd@populate@choice just uses \ekvparse and then gives control to \ekvd@populate@choice@noarg, which throws an error, and \ekvd@populate@choice@.

467 \protected\def\ekvd@populate@choice
468 {%
469 \ekvparse\ekvd@populate@choice@noarg\ekvd@populate@choice@
470 }
471 \protected\long\def\ekvd@populate@choice@noarg#1\%
472 {%
473 \expandafter\ekvd@err@missing@definition@msg\expandafter{\ekvd@cur : #1}%
474 }

\ekvd@populate@choice@ runs the prefix-test, if there is none we can directly define the choice, for that \ekvd@set@choice will expand to the current choice-key’s name, which will have been defined by \ekvd@type@choice. If there is a prefix run the prefix grabbing routine, which was altered for \ekvd@type@choice.

475 \protected\long\def\ekvd@populate@choice@#1\#2\%
476 {%
477 \ekvd@clear@prefixes
478 \expandafter\ekvd@assert@arg@msg\expandafter{\ekvd@cur : #1}%
479 {%
480 \ekvd@ifs\expandafter{\ekvd@assert@arg@msg}\expandafter{\ekvd@cur : #1}%
481 {%
482 \ekvlet\ekvd@set{#1}\ekvd@stop}%
483 }%
Finally we're able to set up the \texttt{@tchoice} macro, which has to store the current choice-key's name, define the key, and parse the available choices.
The unknown type has different subtypes which would be the key names for other types. It is first checked whether that subtype is defined, if it isn’t throw an error, else use that subtype.

```
\protected\long\def\ekvd@type@unknown#1#2\
{\ekv@ifdefined{ekvd@type@unknown@\detokenize{#1}}\
{\csname ekvd@type@unknown@\detokenize{#1}\endcsname#1{#2}}\
\ekvd@err@misused@unknown}
```

The unknown noval type can use `\ekvd@type@unknown@noval` directly (after asserting some prefixes).

```
\protected\long\def\ekvd@type@unknown@noval#1\
{\ekvd@assert@new@for@name{ekv@name\ekvd@set{}uN}\
{\ekvd@assert@arg\
{\ekvd@assert@not@also\
{\ekvd@prot\ekv@defunknownNoVal\ekvd@set{#1}}\
\endcsname#1{#2}}}\
\ekvd@err@misused@unknown}
```

The unknown code type uses some trickery during the definition in order to swap out #1 and #2 in the user supplied definition. This is done via a temporary macro that stores the definition but gets the parameter numbers reversed while the real definition is done.

```
\protected\long\def\ekvd@type@unknown@code#1\
{\ekvd@assert@new@for@name{ekv@name\ekvd@set{}}\uH}\
\ekvd@assert@arg\
{\ekvd@assert@not@also\
{\ekvd@assert@not@long\
{\ekvd@prot\ekv@defunknownNoVal\ekvd@set{#1}}}\
\endcsname#1{#2}}}\
```

(End definition for `\ekvd@type@unknown-choice`)

(End definition for `\ekvd@type@unknown-choice` and others.)
The unknown redirect types also just forward to \texttt{ekvredirectunknown} after asserting some prefixes.

\begin{verbatim}
\protected\edef\ekvd@type@unknown@redirect#1{%
  \expandafter\noexpand\csname ekvd@type@unknown@redirect-code\endcsname{#1}%
  \expandafter\noexpand\csname ekvd@type@unknown@redirect-noval\endcsname{#1}%
}
\protected\expandafter\def\csname ekvd@type@unknown@redirect-code\endcsname#1{%
  \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}u}%
  \ekvd@assert@arg
  \ekvd@assert@not@also
  \ekvd@assert@not@protected
  \expandafter\ekvredirectunknown\expandafter{\ekvd@set}{#1}%
}\%
\protected\expandafter\def\csname ekvd@type@unknown@redirect-noval\endcsname#1{%
  \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}uN}%
  \ekvd@assert@arg
  \ekvd@assert@not@also
  \ekvd@assert@not@protected
  \ekvd@assert@not@long
  \expandafter\ekvredirectunknownNoVal\expandafter{\ekvd@set}{#1}%
}\%
\end{verbatim}

(End definition for \texttt{\ekvd@type@unknown@redirect}, \texttt{\ekvd@type@unknown@redirect-code}, and \texttt{\ekvd@type@unknown@redirect-noval}.)

2.2.2 Key Type Helpers

There are some keys that might need helpers during their execution (not during their definition, which are gathered as \texttt{@type@} macros). These helpers are named \texttt{@h@}.
The choice helper will just test whether the given choice was defined, if not throw an error expandably, else call the macro which stores the code for this choice.

```latex
\def\ekvd@h@choice#1{% 
\expandafter\ekvd@h@choice@
\csname\ifcsname#1\endcsname#1\else \relax\fi\endcsname
{#1}%
}\def\ekvd@h@choice@#1#2{% 
\ifx#1\relax
\ekvd@err@choice@invalid{#2}%
\expandafter\@gobble
\fi
#1%
}
```

(End definition for \ekvd@h@choice and \ekvd@h@choice@.)

### Handling also

```latex
\protected\long\def\ekvd@add@val#1#2#3{% 
\ekvd@assert@val{#1}%
{\expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}\endcsname{{##1}}{#1}{#2}{\ekvd@long\ekvdef}{#3}}%
}
\protected\long\def\ekvd@add@noval#1#2#3{% 
\ekvd@assert@noval{#1}%
{\expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}N\endcsname{}{#1}{#2}\ekvdefNoVal{#3}}%
}
\protected\long\def\ekvd@add@aux#1#2{% 
\ekvd@extract@prefixes#1%
\expandafter\ekvd@add@aux@\expandafter{#1#2}%
}
\protected\long\def\ekvd@add@aux@#1#2#3#4#5{% 
#5%
\ekvd@prot#4\ekvd@set{#2}{}{#1#3}%
}
```

(End definition for \ekvd@add@val and others.)

This macro checks which prefixes were used for the definition of a macro and sets \ekvd@long and \ekvd@prot accordingly.
In the following definition \#1 will get replaced by \texttt{macro:}, \#2 by \texttt{\long} and \#3 by \texttt{\protected} (in each, all tokens will have category other). This allows us to parse the meaning of a macro for those strings.

We use a temporary macro to expand the three arguments of \texttt{\ekvd@extract@prefixes@}, which will set up the real meaning of itself and the parsing for \texttt{\long} and \texttt{\protected}.

These macros test whether a control sequence is defined, if it isn't they define it, either via \texttt{\let} or via the correct \texttt{\new\langle reg\rangle}.

(End definition for \texttt{\ekvd@extract@prefixes@} and others.)

2.2.4 Tests

\texttt{\ekvd@newlet} and \texttt{\ekvd@newreg}
A test for exactly two tokens can be reduced for an empty-test after gobbling two tokens, in the case that there are fewer tokens than two in the argument, only macros will be gobbled that are needed for the true branch, which doesn’t hurt, and if there are more this will not be empty.

\long\def\ekvd@assert@twoargs#1{\ekvd@ifnottwoargs{#1}{\ekvd@err@missing@definition}}
\long\def\ekvd@ifnottwoargs#1{\ekvd@ifempty@gtwo#1\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo}
\long\def\ekvd@ifempty@gtwo#1#2{\ekv@ifempty@}

(End definition for \ekvd@assert@twoargs, \ekvd@ifnottwoargs, and \ekvd@ifempty@gtwo.)

Assert that a given key is defined as a value taking key or a NoVal key with the correct argument structure, respectively.

\protected\def\ekvd@assert@val#1{\ekvifdefined\ekvd@set{#1}{\expandafter\ekvd@assert@val@\csname\ekv@name\ekvd@set{#1}\endcsname}else{\ekvd@assert@noval@\ekvd@set{#1}}}
\protected\def\ekvd@assert@val@#1{\expandafter\ekvd@extract@args\meaning#1\ekvd@stop\unless\ifx\ekvd@extracted@args\ekvd@one@arg@string\ekvd@err@unsupported@arg\fi\@firstofone}
\protected\def\ekvd@assert@noval@#1{\expandafter\ekvd@extract@args\meaning#1\ekvd@stop\unless\ifx\ekvd@extracted@args\ekvd@empty\ekvd@err@unsupported@arg\fi\@firstofone}
\protected\def\ekvd@extract@args#1\%{
  \protected\def\ekvd@extract@args##1#1##2->##3\ekvd@stop
  \def\ekvd@extracted@args{##2}
}\expandafter\ekvd@extract@args\expandafter{\detokenize{macro:}}\edef\ekvd@one@arg@string{\string#1}

(End definition for \ekvd@assert@val and others.)

\ekvd@assert@arg
\ekvd@assert@arg@msg
\ekvd@ifnoarg

\long\def\ekvd@assert@arg{
  \ekvd@ifnoarg\ekvd@err@missing@definition}
\long\def\ekvd@assert@arg@msg#1\%
\ekvd@ifnoarg{\ekvd@err@missing@definition@msg{#1}}

(End definition for \ekvd@assert@arg, \ekvd@assert@arg@msg, and \ekvd@ifnoarg.)

\ekvd@assert@filledarg
\ekvd@ifnoarg@or@empty

\long\def\ekvd@assert@filledarg#1\%
\ekvd@ifnoarg@or@empty{#1}\ekvd@err@missing@definition
\long\def\ekvd@ifnoarg@or@empty#1\%
\ekvd@ifnoarg@or@empty{#1}

(End definition for \ekvd@assert@filledarg and \ekvd@ifnoarg@or@empty.)

\ekvd@assert@not@long
\ekvd@assert@not@protected
\ekvd@assert@not@long@also
\ekvd@assert@not@protected@also
\ekvd@assert@not@new
\ekvd@assert@not@new@for@name

\def\ekvd@assert@not@long{\ifx\ekvd@long\long\ekvd@err@no@prefix{long}\fi}
\def\ekvd@assert@not@protected{\ifx\ekvd@prot\protected\ekvd@err@no@prefix{protected}\fi}
\def\ekvd@assert@not@long@also{\ifx\ekvd@long\long@also{\ekvd@err@no@prefix{long@also}}}
\def\ekvd@assert@not@protected@also{\ifx\ekvd@protected@also{\ekvd@err@no@prefix{protected@also}}}
\def\ekvd@assert@not@new{\ifx\ekvd@new\ekvd@err@no@prefix{new}\fi}
\def\ekvd@assert@not@new@for@name{\ekvd@err@not@new@for@name}
\def\ekvd@assert@not@new@for@name{\ekvd@err@not@new@for@name}

Some key-types don't want to be also, \long or \protected, so we provide macros to test this and throw an error, this could be silently ignored but now users will learn to not use unnecessary stuff which slows the compilation down.

28
It is bad to use also on a key that already contains a choice, as both choices would share the same valid values and thus lead to each callback being used twice. The following is a rudimentary test against this.

\begin{verbatim}
\protected\def\ekvd@if@not@already@choice#1\ekvd@h@choice\ekvd@stop{
\expandafter\ekvd@if@not@already@choice@a
\csname\ekv@name\ekvd@set{#1}\endcsname\ekvd@h@choice\ekvd@stop
}
\protected\def\ekvd@if@not@already@choice@a{
\expandafter\ekvd@if@not@already@choice@b}
\long\protected\def\ekvd@if@not@already@choice@b#1\ekvd@h@choice#2\ekvd@stop{
\ekv@ifempty{#2}\@firstofone\@gobble
}
\end{verbatim}

(End definition for \ekvd@if@not@already@choice, \ekvd@if@not@already@choice@a, and \ekvd@if@not@already@choice@b.)

Yet another test which can be reduced to an if-empty, this time by gobbling everything up to the first space.

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

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

## 2.2.5 Messages

Most messages of \texttt{expkv-def} are not expandable, since they only appear during key-definition, which is not expandable anyway.

The non-expandable error messages are boring, so here they are:

\begin{verbatim}
\protected\def\ekvd@errm#1{\errmessage{expkv-def Error: #1}}
\protected\def\ekvd@err@missing@definition{\ekvd@errm{Missing definition for key \texttt{\ekvd@cur}}}\protected\def\ekvd@err@missing@definition@msg{\ekvd@errm{Missing definition for key \texttt{\ekvd@cur}}}\protected\def\ekvd@err@missing@definition@msg#1{\ekvd@errm{Missing definition for key \texttt{\ekvd@cur}}}\protected\def\ekvd@err@missing@definition@msg#1{\ekvd@errm{Missing definition for key \texttt{\ekvd@cur}}}\protected\def\ekvd@err@missing@definition@msg#1{\ekvd@errm{Missing definition for key \texttt{\ekvd@cur}}}\protected\def\ekvd@err@missing@definition@msg#1{\ekvd@errm{Missing definition for key \texttt{\ekvd@cur}}}\protected\def\ekvd@err@missing@definition@msg#1{\ekvd@errm{Missing definition for key \texttt{\ekvd@cur}}}\protected\def\ekvd@err@missing@definition@msg#1{\ekvd@errm{Missing definition for key \texttt{\ekvd@cur}}}\protected\def\ekvd@err@missing@definition@msg#1{\ekvd@errm{Missing definition for key \texttt{\ekvd@cur}}}\end{verbatim}
The expandable error messages use \ekvd@err, which is just like \ekv@err from expkv. It uses a runaway argument to start the error message.

\ekv@exparg{\long\def\ekvd@err#1}{\ekverr{expkv-def}{#1}}

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

\catcode`@=\ekvd@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.
set .................................................. 6
skip .................................................. 5
smeta .................................................. 6
smeta .................................................. 6
store .................................................. 5

T
\textbf{T}_{\text{eX}} \text{ and } \textbf{T}_{\text{eX}} \text{ x}_{\text{e}} \text{ commands:}
\begin{verbatim}
\texttt{\textbackslash ekvd@assert@not@protected} ................. 77, 180, 596, 608, 762
\texttt{\textbackslash ekvd@assert@not@protected@also} ................. 87, 762
\texttt{\textbackslash ekvd@assert@noval} .................. 638, 702
\texttt{\textbackslash ekvd@assert@noval@} .................. 702
\texttt{\textbackslash ekvd@assert@twoargs} ................. 243, 245, 248, 253, 429, 592
\texttt{\textbackslash ekvd@assert@val} .................. 451, 630, 702
\texttt{\textbackslash ekvd@choice@invalid@p} ................. 702
\texttt{\textbackslash ekvd@choice@invalid@p@ also} ................. 521
\texttt{\textbackslash ekvd@choice@p@long} .................. 444
\texttt{\textbackslash ekvd@choice@p@long@ also} ................. 444
\texttt{\textbackslash ekvd@choice@p@new} .................. 522
\texttt{\textbackslash ekvd@choice@p@protected} ................. 444
\texttt{\textbackslash ekvd@choice@p@protected@ also} ................. 444
\texttt{\textbackslash ekvd@choice@p@prefix} ................. 444
\texttt{\textbackslash ekvd@choice@p@prefix@ also} ................. 444
\texttt{\textbackslash ekvd@clear@prefixes} ................. 20, 46, 477
\texttt{\textbackslash ekvd@cur} ................. 47, 473, 478, 518, 808, 812, 816, 821, 824, 828, 830, 832, 839, 843, 845
\texttt{\textbackslash ekvd@empty} ................. 20, 379, 733
\texttt{\textbackslash ekvd@err} ................. 866, 869
\texttt{\textbackslash ekvd@err@add@noval@on@val} ................. 727, 806
\texttt{\textbackslash ekvd@err@add@noval@on@val@ also} ................. 708, 806
\texttt{\textbackslash ekvd@err@choice@invalid@val} ................. 623, 846
\texttt{\textbackslash ekvd@err@choice@invalid@val@ also} ................. 846
\texttt{\textbackslash ekvd@err@missing@definition} ................. 81, 694, 747, 754, 806
\texttt{\textbackslash ekvd@err@missing@definition@msg} ................. 843
\texttt{\textbackslash ekvd@err@no@prefix} ................. 473, 750, 806
\texttt{\textbackslash ekvd@err@no@prefix@type} ................. 50, 67, 806
\texttt{\textbackslash ekvd@err@usage@unknown} ................. 552, 844
\texttt{\textbackslash ekvd@err@not@defined@key} ................. 762, 764, 765, 773, 806
\texttt{\textbackslash ekvd@err@not@defined@key@ also} ................. 767, 769, 806
\texttt{\textbackslash ekvd@err@not@defined@key@msg} ................. 518, 806
\texttt{\textbackslash ekvd@err@new} ................. 771, 780, 806
\texttt{\textbackslash ekvd@err@new@for@name} ................. 147, 168, 185, 709, 728, 806
\texttt{\textbackslash ekvd@err@undefined@prefix} ................. 60, 507, 806
\texttt{\textbackslash ekvd@err@unsupported@arg} ................. 717, 736, 806
\texttt{\textbackslash ekvd@errm} ................. 506
\texttt{\textbackslash ekvd@extract@args} ................. 702
\end{verbatim}