Abstract

\texttt{expkv\texttt{def}} provides a small \langle key \rangle=\langle value \rangle interface to define keys for \texttt{expkv}. Key-types are declared using prefixes, similar to static typed languages. The stylised name is \texttt{expkv\texttt{def}} but the files use \texttt{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 .................................................. 8
  1.4 Example .............................................. 9
  1.5 License ............................................. 10

2 Implementation 11
  2.1 The \LaTeX Package .................................. 11
  2.2 The Con\TeXt module ................................ 11
  2.3 The Generic Code .................................. 11
    2.3.1 Key Types ...................................... 14
    2.3.2 Key Type Helpers .............................. 28
    2.3.3 Handling also ................................. 29
    2.3.4 Tests ........................................... 30
    2.3.5 Messages ....................................... 33

Index 36

\*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`, go define ⟨key⟩=⟨value⟩ interfaces with ⟨key⟩=⟨value⟩ interfaces.

Unlike many of the other established ⟨key⟩=⟨value⟩ interfaces to define keys, `expkv` 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 for the first space is way easier than parsing for the last one. `expkv`’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` is usable as generic code, as a LaTeX package, and as a ConTeXt module. It’ll automatically load `expkv` in the same mode as well. To use it, just use one of

\begin{verbatim}
\input expkv-def % plain TeX
\usepackage{expkv-def} % LaTeX
\usemodule[expkv-def] % ConTeXt
\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
\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 \langle value \rangle. On the other hand new and also might be simple on first sight as well, but their rules are a bit more complicated.

\texttt{also} The following key type will be \textit{added} to an existing \langle key \rangle’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 \texttt{long} or \texttt{protected} to a key which already is \texttt{long} or \texttt{protected} (the opposite order would be suboptimal as well, but can’t be really captured with the current code).

A key already defined as \texttt{long} or \texttt{protected} will stay \texttt{long} or \texttt{protected}, but you can as well add \texttt{long} or \texttt{protected} with the \texttt{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

\begin{verbatim}
\ekvdefinekeys{also-example}
{
  bool key = \ifmybool,
  also code key = \domystuff[#1]
}
\end{verbatim}

If you use \texttt{also} on a \texttt{choice}, \texttt{bool}, \texttt{invbool}, or \texttt{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.

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

\texttt{long} The following key will be defined \texttt{long}.

\texttt{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. \texttt{new} only asserts that there are no conflicts between \texttt{NoVal} keys and other \texttt{NoVal} keys or value taking keys and other value taking keys. For example you can use the following without an error:

\begin{verbatim}
\ekvdefinekeys{new-example}
{
  code key = \domystuffwitharg[#1],
  new noval key = \domystuffwithoutarg
}
\end{verbatim}

1.2.2 \texttt{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.

```
\texttt{code} \langle \texttt{key} \rangle = \{ \langle \texttt{definition} \rangle \} \texttt{new also protected long}
\texttt{ecode} \langle \texttt{key} \rangle = \{ \langle \texttt{definition} \rangle \} \texttt{new also protected long}
```

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

```
\texttt{noval} \langle \texttt{key} \rangle = \{ \langle \texttt{definition} \rangle \} \texttt{new also protected long}
\texttt{enoval} \langle \texttt{key} \rangle = \{ \langle \texttt{definition} \rangle \} \texttt{new also protected long}
```

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

```
\texttt{default} \langle \texttt{key} \rangle = \{ \langle \texttt{definition} \rangle \} \texttt{new also protected long}
\texttt{qdefault} \langle \texttt{key} \rangle = \{ \langle \texttt{definition} \rangle \} \texttt{new also protected long}
\texttt{odefault} \langle \texttt{key} \rangle = \{ \langle \texttt{definition} \rangle \} \texttt{new also protected long}
\texttt{fdefault} \langle \texttt{key} \rangle = \{ \langle \texttt{definition} \rangle \} \texttt{new also protected long}
```

This serves to place a default \langle \texttt{value} \rangle for a \langle \texttt{key} \rangle that takes an argument, the \langle \texttt{key} \rangle can be of any argument-grabbing kind, and when used without a \langle \texttt{value} \rangle it will be passed \langle \texttt{definition} \rangle instead. The qdefault variant will expand the \langle \texttt{key} \rangle's code once, so will be slightly quicker, but not change if you redefine \langle \texttt{key} \rangle. 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 \langle \texttt{key} \rangle-code with \langle \texttt{definition} \rangle as its argument inside of an \texttt{\edef}.

```
\texttt{initial} \langle \texttt{key} \rangle = \{ \langle \texttt{value} \rangle \} \texttt{new also protected long}
\texttt{oinitial} \langle \texttt{key} \rangle = \{ \langle \texttt{value} \rangle \} \texttt{new also protected long}
\texttt{finitial} \langle \texttt{key} \rangle = \{ \langle \texttt{value} \rangle \} \texttt{new also protected long}
\texttt{einitial} \langle \texttt{key} \rangle = \{ \langle \texttt{value} \rangle \} \texttt{new also protected long}
```

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

If you don't provide a value (and no equals sign) a noval \langle \texttt{key} \rangle of the same name is called once (or, if you specified a default for a value taking key that would be used).
bool \langle key \rangle = \langle cs \rangle

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

invbool

\texttt{invbool} variants will both take two arguments and if true the first will be used else the second, so they are always either \texttt{\@firstoftwo} or \texttt{\@secondoftwo}. The variants with a leading \texttt{g} will set the control sequence globally, the others locally. If \langle cs \rangle is not yet defined it'll be initialised as the \texttt{false} version. Note that the initialisation is \texttt{not} done with \texttt{\newif}, so you will not be able to do \texttt{\iftrue} outside of the \langle key \rangle = \langle value \rangle interface, but you could use \texttt{\newif} yourself. Even if the \langle key \rangle 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 (\textit{e.g.}, you can use \texttt{edefault} \langle key \rangle = \texttt{false} without an issue to change the default behaviour to execute the \texttt{false} choice). Internally a \texttt{bool} \langle key \rangle is the same as a choice key which is set up to handle \texttt{true} and \texttt{false} as choices.

boolpair

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

store
data

gstore
dataT
gdata

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

edata
dataT

gdataT
data

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

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

This key type can set other keys, but this key doesn’t take a \value{}, so the \key{}=\value{} list is static.

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

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

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 set \langle key \rangle = \{\langle key \rangle\}. The created \texttt{set} key will not take a \langle value \rangle. Note that just like in \texttt{\exPkv} it'll not be checked whether \langle set \rangle is defined and you'll get a low-level \TeX error if you use an undefined \langle set \rangle.

choice \texttt{choice \langle key \rangle = \{\langle value \rangle=\langle definition \rangle, \ldots\}} \texttt{\ new \ also \ protected \ long}

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=\langle definition \rangle list. If a defined \langle value \rangle is passed to \langle key \rangle the \langle definition \rangle will be left in the input stream. You can make individual values protected inside the \langle value \rangle=\langle definition \rangle 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.

choice-store \texttt{choice-store \langle key \rangle = \{\langle cs \rangle\langle value \rangle, \ldots\}} \texttt{\ new \ also \ protected \ long}

This is defines a special type of choice key that'll store the given choice inside the macro \langle cs \rangle (so \langle cs \rangle should be a single control sequence name such as \texttt{\foo}). Since storing inside a macro can't be done expandably every choice-code is \texttt{\protected}, you might define the choice-store key itself as \texttt{\protected} as well if you want. Since the definition of each choice is predefined with this key type the choice list should just be a comma separated list of valid choices.

This means that the following choice and choice-store keys are equivalent at use time:

\begin{verbatim}
\newcommand\mya{}
\ekvdefinekeys{example}
  \{ choice key1 = {a=\def\mya{a}, b=\def\mya{b}, c=\def\mya{c}}
  ,choice-store key2 = \mya{a,b,c} \}
\end{verbatim}

choice-enum \texttt{choice-enum \langle key \rangle = \{\langle cs \rangle\langle value \rangle, \ldots\}} \texttt{\ new \ also \ protected \ long}

This is similar to choice-store, the differences are: \langle cs \rangle should be a count-register or is initialised as such if the \langle cs \rangle is undefined (via \texttt{\newcount}); instead of the value the position of the value in the given list is stored in this register (zero-based).

This means that the following choice and choice-enum keys are equivalent at use time:

\begin{verbatim}
\newcount\myb
\ekvdefinekeys{example}
  \{ choice key1 = {a=\myb=0, b=\myb=1, c=\myb=2} 
  ,choice-enum key2 = \myb{a,b,c} \}
\end{verbatim}
\end{verbatim}
unknown-choice

unknown-choice (key) = \{(definition)\}
new also protected long

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)\}
new also protected long

By default \texttt{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 \texttt{\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 \texttt{\ekvdefunknown} directly.}

unknown_noval

unknown noval = \{(definition)\}
new also protected long

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

unknown_redirect-code

unknown redirect-code = \{(set-list)\}
new also protected long

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 \texttt{expkv’s \ekvredirectunknown} will be used.

unknown_redirect-noval

unknown redirect-noval = \{(set-list)\}
new also protected long

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

unknown_redirect

unknown redirect = \{(set-list)\}
new also protected long

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}
{
  long code keyA = #1
  noval 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 = \empty
  new box 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]
{\%
  \begingroup
  \ekvset{ex}{#1}%
  \begin{description}
  \item{\Pname} {errmessage{A person requires a name}}
  \item{Age} \Page{\textit}{errmessage{A person requires an age}}
  \item{Hobbies}
  \end{description}
  \%\begingroup
}

9
\end{description}
\end{group}
}\Persona{name=Jonathan P. Spratte, age=young, hobby=\TeX\ coding}
\Persona{name=Some User, age=unknown, hobby=Reading Documentation}
\Persona{name=Anybody, age=any}

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:

Jonathan P. Spratte
\begin{itemize}
\item Age young
\item Hobbies \TeX\ coding
\end{itemize}

Some User
\begin{itemize}
\item Age unknown
\item Hobbies Reading Documentation
\end{itemize}

Anybody
\begin{itemize}
\item Age any
\end{itemize}

1.5 License

Copyright © 2020–2022 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\ekvd@tmp{
  \ProvidesFile{expkv-def.tex}%
    \[ekvdDate\space v\ekvdVersion\space a key-defining frontend for expkv]%
}
\input{expkv-def.tex}
\ProvidesPackage{expkv-def}%
    \[ekvdDate\space v\ekvdVersion\space a key-defining frontend for expkv]
\end{verbatim}

2.2 The \texttt{ConTeXt} module

\begin{verbatim}
\writestatus{loading}{ConTeXt User Module / expkv-def}
\usemodule[expkv]
\unprotect
\input expkv-def.tex
\writestatus{loading}{ConTeXt User Module / expkv-def / Version \ekvdVersion\space loaded}
\protect\endinput
\end{verbatim}

2.3 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

\expandafter\ifx\csname ekvdVersion\endcsname\relax
\else
\expandafter\endinput
\fi

\end{verbatim}

We’re on our first input, so let’s store the version and date in a macro.

\begin{verbatim}
def\ekvdVersion{0.9}
def\ekvdDate{2022-01-29}
\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}
\csname ekvd@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 ekvd@tmp\endcsname=\texttt{\@}=11
\end{verbatim}
\texttt{\textbackslash 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{\textbackslash expKVdef} was initialized.

\texttt{\textbackslash ekvd@ifprimitive}

\begin{verbatim}
\protected\long\def\ekvd@ifprimitive#1\protect\%\endverbatim

\texttt{\textbackslash ifprimitive} will use \texttt{\textbackslash ekvd@long}, \texttt{\textbackslash ekvd@prot}, and \texttt{\textbackslash ekvd@ifalso} to store whether a key should be defined as \texttt{\textbackslash long} or \texttt{\textbackslash protected} or adds an action to an existing key, and we have to clear them for every new key. By default \texttt{\textbackslash long} and \texttt{\textbackslash protected} will just be empty, \texttt{\textbackslash ifalso} will be \texttt{\textbackslash @secondoftwo}, and \texttt{\textbackslash ifnew} will just use its third argument.

\texttt{\textbackslash ekvd@clear@prefixes}

\begin{verbatim}
\protected\def\ekvd@clear@prefixes\protect\{}\end{verbatim}

\texttt{\textbackslash clear@prefixes} (End definition for \texttt{\textbackslash ekvd@clear@prefixes}.

\texttt{\textbackslash ekvdefinekeys}

This is the one front-facing macro which provides the interface to define keys. It’s using \texttt{\textbackslash ekvparse} to handle the \texttt{\langle key\rangle=\langle value\rangle} list, the interpretation will be done by \texttt{\textbackslash ekvd@noarg} and \texttt{\textbackslash ekvd@arg}. The \texttt{\langle set\rangle} for which the keys should be defined is stored in \texttt{\textbackslash ekvd@set}.

\texttt{\textbackslash ekvd@noarg}

\begin{verbatim}
\protected\long\def\ekvd@noarg#1\protect\%\end{verbatim}

\texttt{\textbackslash noarg} and \texttt{\textbackslash arg} store whether there was a value in the \texttt{\langle key\rangle=\langle value\rangle} pair. \texttt{\textbackslash 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.

\texttt{\textbackslash ekvd@handle}

\begin{verbatim}
\protected\long\def\ekvd@handle\protect\%\end{verbatim}

\texttt{\textbackslash handle} is documented on page \pageref{ekvdefinekeys}. This function is documented on page \pageref{ekvdefinekeys}.

\texttt{\textbackslash ekvd@noarg}

\begin{verbatim}
\protected\long\def\ekvd@noarg\protect\%\end{verbatim}

\texttt{\textbackslash noarg}
\protected\long\def\ekvd@noarg#1\% 
\let\ekvd@ifnoarg\@secondoftwo
\expandafter\ekvd@handle\detokenize{#1}\ekvd@stop
\protected\long\def\ekvd@arg#1\% 
\let\ekvd@ifnoarg\@secondoftwo
\expandafter\ekvd@handle\detokenize{#1}\ekvd@stop
\protected\long\def\ekvd@handle#1\ekvd@stop#2\% 
\ekvd@clear@prefixes
\def\ekvd@cur{#1}\%
\ekvd@ifspace{#1}\%
{\ekvd@prefix\ekv@mark#1\ekv@stop{#2}}%
\ekvd@err@missing@type
(End definition for \ekvd@noarg, \ekvd@arg, and \ekvd@handle.)
\ekvd@prefix
\expandafter separates prefixes into two groups, the first being prefixes in the \TeX{} sense (long and protected) which use \texttt{@p@} in their name, the other being key-types (\texttt{code}, \texttt{int}, etc.) which use \texttt{@t@} instead. \ekvd@prefix 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{#1}\ekvd@prefix@\ekv@mark\}
\protected\def\ekvd@prefix@#1#2\ekv@stop 
\ekv@ifdefined{ekvd@t@#1}\%
{\ekv@strip{#2}{\csname ekvd@t@#1@endcsname}}%
\ekv@ifdefined{ekvd@p@#1}\%
{\csname ekvd@p@#1@endcsname}\ekvd@prefix@after@p{#2}%
\ekvd@err@undefined@prefix{#1}\@gobble%
%
(End definition for \ekvd@prefix and \ekvd@prefix@.)
\ekvd@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%
%
(End definition for \ekvd@prefix@after@p.)
\ekvd@p@long 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\}

(End definition for \ekvd@p@long and others.)
2.3.1 Key Types

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

\begin{verbatim}
\protected\def\ekvd@type@set#1#2\protect\%{
\ekvd@assert@not@long
\ekvd@assert@not@protected
\ekvd@ifnew{NoVal}{#1}\protect\%{
\ekv@ifempty{#2}\protect\%
{\ekvd@err@missing@definition}\protect\%
{\ekvd@ifalso{
\ekv@expargtwice{\ekvd@add@noval{#1}}
}{\ekvchangeset{#2}}}
\ekvd@assert@not@protected@also
}{\ekv@expargtwice{\ekvdefNoVal\ekvd@set{#1}}{\ekvchangeset{#2}}}\protect\%
}
\end{verbatim}

(End definition for \ekvd@type@set and \ekvd@t@set.)

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.

\begin{verbatim}
\protected\long\def\ekvd@type@noval#1#2#3\protect\%{
\ekvd@ifnew{NoVal}{#2}\protect\%
{\ekvd@ifnoarg\protect\%
{\ekvd@assert@arg\protect\%
{\ekvd@assert@not@long
\prot#1\ekvd@tmp{#3}\protect\%
\ekvd@ifalso\protect\%
{\ekv@exparg{\ekvd@add@noval{#1}}\ekvd@tmp{}}\protect\%
{\ekvletNoVal\ekvd@set{#2}\ekvd@tmp{}}\protect\%
\ekvd@assert@not@protected\%
}\protect\%
{\ekv@expargtwice{\ekvd@add@noval{#1}}}\protect\%
{\ekvchangeset{#2}}\protect\%
\ekvd@assert@not@protected@also\protect\%
}{\ekv@expargtwice{\ekvdefNoVal\ekvd@set{#1}}}\protect\%
{\ekvchangeset{#2}}}\protect\%
}
\end{verbatim}

(End definition for \ekvd@type@noval, \ekvd@t@noval, and \ekvd@t@enoval.)
\ekvd@type@code
code is simple as well, \ekvd@code has to use \edef on a temporary macro, since exp\a does
\text{\textbackslash}doesn't provide an \ekvd@edef.
\begin{verbatim}
\protected\long\def\ekvd@type@code#1#2#3%{
  \ekvd@ifnew{}{#2}\
  \ekvd@assert@arg
  \ekvd@prot\ekvd@long#1\ekvd@tmp##1{#3}%
  \ekvd@ifalso
    \ekv@exparg{\ekvd@add@val{#2}}{\ekvd@tmp{##1}}{}
  \else
    \ekvlet\ekvd@set{#2}\ekvd@tmp%
  \fi
}
\protected\def\ekvd@t@code\ekvd@type@code\def
\protected\def\ekvd@t@ecode\ekvd@type@code\edef
\end{verbatim}
(End definition for \ekvd@type@code, \ekvd@t@code, and \ekvd@t@ecode.)

\ekvd@type@default
\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
\text{\textbackslash}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
  \ekvifdefined\ekvd@set{#3}%
    \ekvd@assert@not@new
    \ekvd@assert@not@long
    \ekvd@prot\edef\ekvd@tmp
      \ekv@unexpanded\expandafter\#1%
    \endcsname\ekv@name\ekvd@set{#3}\endcsname
    \ekvd@ifalso
      \ekv@exparg{\ekvd@add@noval{#3}}\ekvd@tmp{}
    \else
      \ekvletNoVal\ekvd@set{#3}\ekvd@tmp%
    \fi
  \else
    \ekvd@err@undefined@key{#3}%
  \fi
}
\protected\def\ekvd@t@default\ekvd@type@default{}
\protected\def\ekvd@t@qdefault\ekvd@type@default{\expandafter\expandafter}
\let\ekvd@t@fdefault\ekvd@t@qdefault
\protected\def\ekvd@t@fdefault\ekvd@type@default{}{\romannumeral'\^^@}
\end{verbatim}
(End definition for \ekvd@type@default and others.)

\ekvd@t@edefault
edefault is too different from default and qdefault to reuse the \texttt{\@type@} macro, as it
doesn't need \unexpanded inside of \edef.
The boolean types are a quicker version of a choice that accept \texttt{true} and \texttt{false}, and set up the \texttt{NoVal} action to be identical to \texttt{\langle key\rangle =true}. The \texttt{true} and \texttt{false} actions are always just \texttt{\let}ting the macro in \#7 to some other macro (e.g., \texttt{\iftrue}).
The boolean pair types are essentially the same as the boolean types, but set two macros instead of one.
\protected\def\ekvd@t@boolpair#1#2% 
{\ekvd@assert@twoargs{#2}\iftrue\iffalse{#1}#2}
\protected\def\ekvd@t@boolpairTF#1#2% 
{\ekvd@assert@twoargs{#2}\iftf{#1}#2}
\protected\def\ekvd@t@gboolpair#1#2% 
{\ekvd@assert@twoargs{#2}\global\iftf{#1}#2}
\protected\def\ekvd@t@boolpairTF#1#2% 
{\ekvd@assert@twoargs{#2}\iftf{#1}#2}

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

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

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

\ekvd@type@box
\ekvd@t@box
\ekvd@t@gbox

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.
\texttt{\textbackslash ekvd@newreg} is a small wrapper which tests whether the first argument is defined and if not does \texttt{\textbackslash csname new#2\textbackslash endcsname#1}.

\begin{verbatim}
\protected\def\ekvd@type@box#1#2#3% {
  \ekvd@ifnew{}{#2}%
  {%\ekvd@assert@filledarg{#3}%
    \ekvd@newreg{box}{#1}%
  \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{verbatim}

(End definition for \texttt{\ekvd@type@box}, \texttt{\ekvd@t@box}, and \texttt{\ekvd@t@gbox}.)

Similar to box, but set the toks.

\begin{verbatim}
\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{toks}{#1}%
  \ekvd@ifalso
    {%\let\ekvd@prot\protected
      \ekvd@add@val{#2}{#1#3={##1}}{}}%
    {\protected\ekvd@long\ekvdef\ekvd@set{#2}{#1#3={##1}}}%
  }%
}%
\protected\def\ekvd@t@toks{\ekvd@type@toks{}}
\protected\def\ekvd@t@gtoks{\ekvd@type@toks{\global}}
\end{verbatim}

(End definition for \texttt{\ekvd@type@toks}, \texttt{\ekvd@t@toks}, and \texttt{\ekvd@t@gtoks}.)

Just like toks, but expand the current contents of the toks register to append the new contents.

\begin{verbatim}
\ekvd@ifprimitive\toksapp
  {%\protected\def\ekvd@type@preapptoks#1#2#3%}
\end{verbatim}
\ekvd@ifnew{}{#2}%
  \ekvd@assert@filledarg[#3]%
  \%\ekvd@neureg#3{toks}%
  \ekvd@ifalso
    \%\let\ekvd@prot\protected
    \ekvd@add@val{#2}{#1#3{#1}}{}
  \%
  \{\protected\ekvd@long\ekvdef\ekvd@set{#2}{#1#3{#1}}\%
  \%
  \}
  \%\protected\def\ekvd@t@apptoks{\ekvd@type@preapptoks\toksapp}
  \%\protected\def\ekvd@t@gapptoks{\ekvd@type@preapptoks\gtoksapp}
  \%\protected\def\ekvd@t@pretoks{\ekvd@type@preapptoks\tokspre}
  \%\protected\def\ekvd@t@gpretoks{\ekvd@type@preapptoks\gtokspre}
\}
\%\protected\def\ekvd@type@apptoks#1#2#3%
  {\%\ekvd@ifnew{}{#2}%
    \%\ekvd@assert@filledarg[#3]%
    \%\ekvd@neureg#3{toks}%
    \ekvd@ifalso
      \%\let\ekvd@prot\protected
      \ekvd@add@val{#2}{#1#3{#1}}{}
    \%
    \{\protected\ekvd@long\ekvdef\ekvd@set{#2}{#1#3{#1}}\%
    \%
    \}
  }
\%\protected\def\ekvd@t@apptoks{}%
The \texttt{\ekvd@type@reg} can handle all the types for which the assignment will just be \texttt{\langle register\rangle = \langle value\rangle}.

\begin{verbatim}
\protected\def\ekvd@t@reg\{\ekvd@type@reg\{\texttt{\langle register\rangle = \langle value\rangle}\}}
\end{verbatim}

\begin{verbatim}
\protected\def\ekvd@t@int\{\ekvd@type@reg\{\texttt{count}\}}
\protected\def\ekvd@t@eint\{\ekvd@type@reg\{\texttt{count}\}\numexpr\relax\}
\protected\def\ekvd@t@gint\{\ekvd@type@reg\{\texttt{count}\}\global\}
\protected\def\ekvd@t@xint\{\ekvd@type@reg\{\texttt{count}\}\global\numexpr\relax\}
\protected\def\ekvd@t@dimen\{\ekvd@type@reg\{\texttt{dimen}\}}
\protected\def\ekvd@t@edimen\{\ekvd@type@reg\{\texttt{dimen}\}\dimexpr\relax\}
\protected\def\ekvd@t@gdimen\{\ekvd@type@reg\{\texttt{dimen}\}\global\}
\protected\def\ekvd@t@xdimen\{\ekvd@type@reg\{\texttt{dimen}\}\global\dimexpr\relax\}
\protected\def\ekvd@t@skip\{\ekvd@type@reg\{\texttt{skip}\}}
\protected\def\ekvd@t@eskip\{\ekvd@type@reg\{\texttt{skip}\}\glueexpr\relax\}
\protected\def\ekvd@t@gskip\{\ekvd@type@reg\{\texttt{skip}\}\global\}
\protected\def\ekvd@t@xskip\{\ekvd@type@reg\{\texttt{skip}\}\global\glueexpr\relax\}
\end{verbatim}

\textit{(End definition for \texttt{\ekvd@type@reg} and others.)}
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\ekv@empty
 \ekvd@ifalso
{\let\ekvd@prot\protected
 \ekvd@add@val{\#3}{\#1\#4{\#2}}\%
}
{\protected\ekvd@long\ekvdef\ekvd@set{\#3}{\#1\#4{\#2}}}%
}
}

\protected\def\ekvd@t@store{\ekvd@type@store\edef{\ekv@unexpanded{##1}}}
\protected\def\ekvd@t@gstore{\ekvd@type@store\xdef{\ekv@unexpanded{##1}}}
\protected\def\ekvd@t@estore{\ekvd@type@store\edef{##1}}
\protected\def\ekvd@t@xstore{\ekvd@type@store\xdef{##1}}

(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{\ekvset{\#1}{\#2}}\%
 \ekvd@ifalso
{\csname ekvlet#1\endcsname\ekvd@set{\#6}\ekvd@tmp}\%
}
}

\protected\long\def\ekvd@type@meta@a\#1\#2\%
{\expandafter\ekvd@type@meta@b\expandafter{\ekvset{\#1}{\#2}}}

\protected\def\ekvd@type@meta@b
{\expandafter\ekvd@type@meta@c}

\protected\long\def\ekvd@type@meta@c\#1\#2\%
{\ekvd@prot\expandafter\ekvd@tmp\%}

\protected\def\ekvd@t@meta{\ekvd@type@meta{}{##1}\ekvd@add@val{##1}{}
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 @p@ type 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.

\protected\def\ekvd@type@choice{\ekvd@assert@not@long\ekv@expargtwice{\ekvd@prot\def\ekvd@tmp##1}}%  
  \ekvd@if@not@already@choice{#1}{%  
    \ekv@exparg\expandafter\expandafter\expandafter{\expandafter\ekvd@choice@name\expandafter{\ekvd@set}{#1}{##1}}%  
  \ekvd@assert@val{#1}{%  
    \ekvd@if\not\ekvd@assert@not@long{\ekvd@tmp}{\ekvd@set{#6}\ekvd@tmp}%  

(End definition for \ekvd@type@choice and others.)
\expandafter\ekvd@add@aux
\csname ekv@name\ekvd@set{#1}\endcsname{##1}{#1}%
\{\ekvd@tmp{##1}\}
\{\ekvdef\ekvd@assert@not@long@also\}
\{\ekvlet\ekvd@set{#1}\ekvd@tmp\}
\ekvd@populate@choice
just uses \ekvparse and then gives control to \ekvd@populate@choice@noarg, which throws an error, and \ekvd@populate@choice@.
\protected\def\ekvd@populate@choice
{\ekvparse\ekvd@populate@choice@noarg\ekvd@populate@choice@}
\protected\long\def\ekvd@populate@choice@noarg#1\
{\expandafter\ekvd@err@missing@definition\expandafter{\ekvd@cur : #1}}
\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@set@choice. If there is a prefix run the prefix grabbing routine, which was altered for @type@choice.
\protected\long\def\ekvd@set@choice@#1#2\
{\ekvd@clear@prefixes
\ekvd@ifspace{#1}\
{\ekvd@choice@prefix\ekv@mark#1\ekv@mark#1\ekv@stop}\
{\expandafter\edef
\csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname
\unexpanded{#2}}}\
\protected\def\ekvd@choice@prefix#1#2\ekv@stop\
{\ekv@strip{#2}{\ekvd@choice@prefix@{#1}}\ekv@mark}
\protected\def\ekvd@choice@prefix@#1#2#3\ekv@stop\
{\ekv@ifdefined{ekvd@choice@p@#2}\
{\csname ekvd@choice@p@#2\endcsname
\ekvd@ifspace{#3}\
{\ekvd@choice@prefix{#3}#3\ekv@stop}\ekvd@choice@prefix@done{#3}}\
{\ekvd@choice@prefix@done{#1}}}\
\protected\def\ekvd@choice@prefix@done#1\%
Finally we’re able to set up the \texttt{\ekv@t@choice} macro, which has to store the current choice-key’s name, define the key, and parse the available choices.

\begin{verbatim}
\protected\long\def\ekvd@t@choice#1#2{\
\ekvd@ifnew{}{#1}{% 
\ekvd@assert@arg{\ekvd@type@choice{#1}}{\ekvd@set@choice{#1}}{\ekvd@populate@choice{#2}}{#2}}}\
\end{verbatim}

(End definition for \texttt{\ekvd@type@choice} and others.)

These two types define a special kind of choice key and are quite similar, the only difference is what the different choices do (hence they use a shared initialisation which differs in the chosen populate step).

\begin{verbatim}
\protected\long\expandafter\def\csname ekvd@t@choice-store\endcsname {\ekvd@type@choicespecial\ekvd@populate@choicestore}\
\protected\long\expandafter\def\csname ekvd@t@choice-enum\endcsname {\ekvd@type@choicespecial\ekvd@populate@choiceenum}
\end{verbatim}

Initialise similar to a choice key. The difference is that we require two arguments (which we assert), a macro to store things in, and a \texttt{csv}-list containing the allowed values. \texttt{#1} is the populate macro according to the type used.

\begin{verbatim}
\protected\long\def\ekvd@type@choicespecial#1#2#3{\
\ekvd@ifnew{}{#2}{% 
\ekvd@assert@twoargs{\ekvd@type@choice{#1}}{\ekvd@set@choice{#1}}{\ekvd@populate@choice{#2}}{#3}}\
\end{verbatim}

25
We initialise the storing macro if it doesn’t yet exist, and then we loop over the value list. The \edefs with \unexpanded are both necessary to be able to store macro parameter tokens (the outer protects at define time, the inner at use time).

This is similar to the population of a choice-store type, but instead of storing the values in a macro this initialises a count and stores the position of the value in the list inside that count (zero-indexed). The space is necessary to terminate the number scanning, which is the reason we use \@firstofone (so that the space after the macro name isn’t gobbled by \TeX).
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.

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

The unknown noval type can use \ekvdefunknownNoVal directly (after asserting some prefixes).

```latex
\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@assert@not@long
\ekvd@prot\ekvdefunknownNoVal\ekvd@set{#1}\%
}
}
}
```

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.

```latex
\protected\long\def\ekvd@type@unknown@code#1\%
{\ekvd@assert@new@for@name{\ekv@name\ekvd@set{}u}\%
{\ekvd@assert@arg
{\ekvd@assert@not@also
{\begingroup
\def\ekvd@tmp##1##2{#1}\%
\ekv@exparg
{\endgroup
\ekvd@prot\ekvd@long\ekvdefunknown\ekvd@set{}
{\ekvd@tmp{##2}{##1}}\%
}

(End definition for \ekvd@t@unknown-choice.)

(End definition for \ekvd@t@unknown, \ekvd@type@unknown@code, and \ekvd@type@unknown@noval.)
The unknown redirect types also just forward to `\ekvredirectunknown` after asserting some prefixes.

```latex
\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 definition for `\ekvd@type@unknown@redirect`, `\ekvd@type@unknown@redirect-code`, and `\ekvd@type@unknown@redirect-noval`.)
```

### 2.3.2 Key Type Helpers

There are some keys that might need helpers during their execution (not during their definition, which are gathered as `@type@` macros). These helpers are named `@h@`.

```latex
\ekvd@h@choice
\ekvd@h@choice0

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@choice0\csname ifcsname#1\endcsname\else relax\fi\endcsname
}{#1}%
\def\ekvd@h@choice0#1#2{%
  \if#1relax
    \ekvd@err@choice@invalid#2%
  \else\gobble\fi
}%
```

(End definition for `\ekvd@h@choice`, `\ekvd@h@choice0`.)

28
2.3.3 Handling also

This macro checks which prefixes were used for the definition of a macro and sets \ekvd@long and \ekvd@prot accordingly:

\verbatim
\protected\def\ekvd@extract@prefixes#1#2#3%{\%\ekvd@extract@prefixes@\meaning#1\ekvd@stop}{\%\expandafter\ekvd@extract@prefixes@\expandafter{#1#2}\ekvd@stop}{\%\expandafter\ekvd@extract@prefixes@long\ekvd@stop}{\let\ekvd@long\long}{\let\ekvd@prot\long}
\end{verbatim}

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 \texttt{meaning} of a macro for those strings.

\verbatim
\protected\def\ekvd@extract@prefixes@#1#2#3%{\%\ekvd@extract@prefixes@0\ekvd@stop}{\%\expandafter\ekvd@extract@prefixes@0\meaning#1\ekvd@stop}{\%\ekvd@extract@prefixes@0\long}{\let\ekvd@mark\@firstofone\ekvd@stop}{\let\ekvd@mark\@gobble\ekvd@stop}{\let\ekvd@long\long}
\end{verbatim}
We use a temporary macro to expand the three arguments of `\ekvd@extract@prefixes@prot`, which will set up the real meaning of itself and the parsing for `\long` and `\protected`.

\begingroup
\edef\ekvd@tmp
{\detokenize{macro:}}
{\string\long}
{\string\protected}
\ekvd@tmp
\(\text{(End definition for `\ekvd@extract@prefixes@` and others.)}\)

2.3.4 Tests

These macros test whether a control sequence is defined, if it isn’t they define it, either via `\let` or via the correct `\new⟨reg⟩`.

\begingroup
\edef\ekvd@tmp
{\detokenize{macro:}}
\endgroup
\ekvd@assert@twoargs
\ekvd@ifnottwoargs
\ekvd@ifempty@gtwo
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.

\begin{verbatim}
\protected\def\ekvd@assert@twoargs#1\%
{\ekvd@ifnottwoargs{#1}{\ekvd@err@missing@definition}\%
 \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\let#1#2}\%
 }
\protected\def\ekvd@ifnottwoargs#1\%
{\ifdefined#1\ekv@fi@gobble\fi\@firstofone{\csname new#2\endcsname#1}\%
 }
\end{verbatim}

\(\text{(End definition for `\ekvd@assert@twoargs` and `\ekvd@newreg`.)}\)
\ekvd@assert@val\ekvd@assert@val\ekvd@assert@val\ekvd@extract@args\ekvd@one@arg@string

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@include\csname\ekv@name\ekvd@set{#1}\endcsname\}\\ekvifdefinedNoVal\ekvd@set{#1}%\ekvd@err@add@val@on@noval\\{\ekvd@err@undefined@key{#1}\}%%\gobble\}\}
\protected\def\ekvd@assert@val@#1{%\expandafter\ekvd@include\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{%\ekvifdefinedNoVal\ekvd@set{#1}%\expandafter\ekvd@include\csname\ekv@name\ekvd@set{#1}N\endcsname\\ekvd@err@unsupported@arg\@firstofone\}
\protected\def\ekvd@assert@noval@#1{%\expandafter\ekvd@include\meaning#1\ekvd@stop\unless\ifx\ekvd@extracted@args\ekv@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@include\expandafter{\detokenize{macro:}}\edef\ekvd@one@arg@string{\string#1}

(End definition for \ekvd@include twoargs, \ekvd@includefnottwoargs, and \ekvd@includeempty@getwo.)

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

There is no need to actually define \ekvd@includefnoarg here, as it will be set by either \ekvd@includearg or \ekvd@includearg.
Some key-types don’t want to be also, \texttt{\long} or \texttt{\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.

\begin{Verbatim}
\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@also{\ekvd@ifalso{\ekvd@err@no@prefix{also}}{}\fi}
\def\ekvd@assert@not@long@also{\ifx\ekvd@long\long\ekvd@err@no@prefix@also{long}\fi}
\def\ekvd@assert@not@protected@also{\ifx\ekvd@prot\protected\ekvd@err@no@prefix@also{protected}\fi}
\def\ekvd@assert@new#1#2%{\csname ekvifdefined#1\endcsname\ekvd@set{#2}{\ekvd@err@not@new}}
\def\ekvd@assert@not@new{\ifx\ekvd@ifnew\ekvd@assert@new\ekvd@err@no@prefix{new}\fi}
\def\ekvd@assert@new@for@name#1%{\ifx\ekvd@ifnew\ekvd@assert@new\ekv@fi@firstoftwo\@secondoftwo{\ekv@ifdefined{#1}\ekvd@err@not@new}\@firstofone}\fi}
\end{Verbatim}

(End definition for \texttt{\ekvd@assert@not@long} and others.)

It is bad to use \texttt{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@ifnot@already@choice{\ekvd@ifnot@already@choicea\ekvd@ifnot@already@choiceb}
\end{Verbatim}

(End definition for \texttt{\ekvd@assert@not@already@choice} and \texttt{\ekvd@ifnot@already@choice}.)
\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 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.
\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 definition for \ekvd@ifspace and \ekvd@ifspace@.)

2.3.5 Messages
Most messages of \expkvdef 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:
\protected\def\ekvd@errm#1{\errmessage{expkv-def Error: #1}}
\protected\def\ekvd@err@missing@definition
{\ekvd@errm{Missing definition for key ‘\ekvd@cur’}}
\protected\def\ekvd@err@missing@definition@msg
{\ekvd@errm{Missing definition for key ‘\ekv@unexpanded{#1}’}}
\protected\def\ekvd@err@missing@type
{\ekvd@errm{Missing type prefix for key ‘\ekvd@cur’}}
\protected\def\ekvd@err@undefined@prefix#1%
{\ekvd@errm{Undefined prefix ‘\ekv@unexpanded{#1}’ found while processing ‘\ekvd@cur’}}
\protected\def\ekvd@err@undefined@key#1%
{\ekvd@errm{Undefined key ‘\ekv@unexpanded{#1}’ found while processing ‘\ekvd@cur’}}

The expandable error messages use `\ekv@err`, which is just like `\ekv@err` from `expkv`. It uses a runaway argument to start the error message.

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

(End definition for `\ekv@err`.)

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

\catcode`@=\ekv@tmp
Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

```
A
also ........................................ 3
apptoks ...................................  6

B
bool .........................................  5
boolpair .....................................  5
boolpairTF ................................  5
boolTF .......................................  5
box ..........................................  6

code .........................................  4
choice .......................................  7
choice-enum ................................  7
choice-store ................................  7
data ..........................................  5
dataT ..........................................  5
default .......................................  4
dimen .........................................  6
dimen .........................................  2

C
ecode .........................................  4
edata ..........................................  5
edataT .......................................  5
edefault .....................................  4
dimen .........................................  6
einital .......................................  4
eint ..........................................  6

D
data ..........................................  5
dataT ..........................................  5
default .......................................  4
dimen .........................................  6
ecode .........................................  4
edata ..........................................  5
edataT .......................................  5
edefault .....................................  4
dimen .........................................  6
einital .......................................  4
eint ..........................................  6

E
ecode .........................................  4
edata ..........................................  5
edataT .......................................  5
edefault .....................................  4
dimen .........................................  6
einital .......................................  4
eint ..........................................  6
\ekvchangeset ................................ 104,107
\ekvcsvloop ................................ 638,650
\ekvdef .....................................  2, 5, 9, 22
\ekvdef ....................................  292, 323, 343, 363, 386, 413, 437, 465, 546, 757
\ekvdefkeys ................................  2, 47
\ekvdefNoVal ................................ 107, 226, 255, 765
\ekvdefunkown ...............................  702
\ekvdefunkownNoVal .........................  686
\ekvdVersion ................................ 2, 5, 9, 15, 22
\ekverr ..................................... 1004
\ekvifdef ..................................  152, 176, 202, 828, 850
\ekvifdefNoVal ................................ 197, 831, 847
\ekvlet ......................................  142, 550
\ekvletNoVal ................................ 127, 163, 184
\ekvpars ....................................  50, 554
\ekvredirectunknown ....................... 721
\ekvredirectunknownNoVal .................. 734
\ekvset .....................................  489
enoval ........................................  4
eskip ..........................................  6
estore ........................................  5
fdefault ......................................  4
finitial .......................................  4

G
gapptoks .....................................  6
gbool ..........................................  5
gboolpair ....................................  5
gboolpairTF ................................  5
gboolTF .......................................  5
gbox ..........................................  6
gdata ..........................................  5
gdataT .......................................  5
gdimen ........................................  6
gint ..........................................  6
ginvbool .....................................  5
ginvboolTF ................................  5
gpretoks ......................................  6
gskip ..........................................  6
gstore ........................................  5
gtoks .........................................  6
\gtoksapp ................................... 368
\gtokspre ..................................  370

H
initial .......................................  4
int ...........................................  6
invbool .....................................  5
invboolTF ...................................  5

I
long ..........................................  3

M
meta ..........................................  6

N
ew ..........................................  3
\newtoks ....................................  394
mmeta ........................................  6
\noexpand ................................... 710, 711
noval ........................................  4
```
toks .......................... 6 \usemodule ....................... 11
\toksapp .......................... 349, 367 \writestatus .................. 10, 14
\tokspre .......................... 369

\unprotect ...................... 12

U
unknown_code ..................... 8 xdata ......................... 5
unknown_noval .................... 8 xdataT ..................... 5
unknown_redirect ................ 8 xdimen ..................... 6
unknown_redirect-code .......... 8 xint ......................... 6
unknown_redirect-noval ........ 8 xskip ....................... 6
unknown-choice .................. 8 xstore ..................... 5