Abstract

\texttt{expkv\defdef} 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\defdef} but the files use \texttt{expkv-def}, this is due to CTAN-rules which don’t allow \textpipe 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 ...................................................... 5
1.4 Example .................................................. 5
1.5 License .................................................. 6

2 Implementation 7
2.1 The \LaTeX\ Package ...................................... 7
2.2 The Generic Code ....................................... 7
2.2.1 Key Types ............................................. 9
2.2.2 Key Type Helpers ................................... 16
2.2.3 Tests ............................................... 16
2.2.4 Messages .......................................... 18

Index 21

\*jspratte@yahoo.de
1 Documentation

Since the trend for the last couple of years goes to defining keys for a \( \langle \text{key} \rangle = \langle \text{value} \rangle \) interface using a \( \langle \text{key} \rangle = \langle \text{value} \rangle \) interface, I thought that maybe providing such an interface for \texttt{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 \texttt{expkv}’s initial scope. So here it is \texttt{expkv\_def}, go define \( \langle \text{key} \rangle = \langle \text{value} \rangle \) interfaces with \( \langle \text{key} \rangle = \langle \text{value} \rangle \) interfaces.

Unlike many of the other established \( \langle \text{key} \rangle = \langle \text{value} \rangle \) interfaces to define keys, \texttt{expkv\_def} works using prefixes instead of suffixes (e.g., \texttt{.tl\_set:N} of \texttt{l3keys}) or directory like handlers (e.g., \texttt{/store in} of \texttt{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. \texttt{expkv\_def}’s prefixes are sorted into two categories: \texttt{p}-type, which are equivalent to \TeX{}’s prefixes like \texttt{\long}, and \texttt{t}-type defining the type of the key. For a description of the available \texttt{p}-prefixes take a look at \texttt{subsubsection 1.2.1}, the \texttt{t}-prefixes are described in \texttt{subsubsection 1.2.2}.

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

\begin{verbatim}
\usepackage{expkv-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\{\langle set\rangle\}\{\langle key\rangle=\langle value\rangle, \ldots\}
\end{verbatim}

In \( \langle set\rangle \), define \( \langle key\rangle \) to have definition \( \langle value\rangle \). The general syntax for \( \langle key\rangle \) should be

\begin{verbatim}
\langle prefix\rangle \langle name\rangle
\end{verbatim}

Where \( \langle prefix\rangle \) is a space separated list of optional \texttt{p}-type prefixes followed by one \texttt{t}-type prefix. The syntax of \( \langle value\rangle \) is dependent on the used \texttt{t}-prefix.

\begin{verbatim}
\ekvDate \ekvVersion
\end{verbatim}

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

1.2 Prefixes

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

1.2.1 \texttt{p}-Prefixes

The \texttt{p}-type prefixes are pretty simple by nature, so their description is pretty simple. They affect the \( \langle key\rangle \) at use-time, so omitting \texttt{long} doesn’t mean that a \texttt{\definition} can’t contain a \texttt{\par} token, only that the \( \langle key\rangle \) will not accept a \texttt{\par} in \( \langle value\rangle \).
The following key will be defined \protected. Note that key-types which can’t be defined expandable will always use \protected.

The following key will be defined \long.

1.2.2 \texttt{t}-Prefixes

Since the \texttt{p}-type prefixes apply to some of the \texttt{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 (\protect), and disallowed prefixes will be red (\protect). This will be put flush-right in the syntax showing line.

code
\texttt{\texttt{code} \langle key \rangle = \{(definition)\}} \hspace{6cm} \text{protected long}

define \langle key \rangle to expand to \langle definition \rangle. The \langle key \rangle will require a \langle value \rangle for which you can use \#1 inside \langle definition \rangle. The ecode variant will fully expand \langle definition \rangle inside an \edef.

noval
\texttt{noval \langle key \rangle = \{(definition)\}} \hspace{6cm} \text{protected long}

The noval type defines \langle key \rangle to expand to \langle definition \rangle. The \langle key \rangle will not take a \langle value \rangle.

edefault
\texttt{edefault \langle key \rangle = \{(definition)\}} \hspace{6cm} \text{protected long}

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

initial
\texttt{initial \langle key \rangle = \{(value)\}} \hspace{6cm} \text{protected long}

With initial you can set an initial \langle value \rangle for an already defined argument taking \langle key \rangle. It’ll just call the key-macro of \langle key \rangle and pass it \langle value \rangle.

\texttt{\texttt{bool} \langle key \rangle = \langle cs \rangle \hspace{6cm} \text{protected long}}

The \langle cs \rangle should be a single control sequence, such as \texttt{\iffoo}. This will define \langle key \rangle to be a boolean key, which only takes the values true or 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=true. bool and gbool will behave like \texttt{\iftrue} or \texttt{\iffalse}. The boolean and 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 not done with \protect, so you will not be able to do \texttt{\footrue} outside of the \langle key \rangle=\langle value \rangle interface, but you could use \protect yourself. Even if the \langle key \rangle will not be \protect the commands which execute the true or false choice will be, so the usage should be safe in an expansion context (e.g., you can use \edefualt{\langle key \rangle}=\texttt{false} without an issue to change the default behaviour to execute the \texttt{false} choice).
store \langle key \rangle = \langle cs \rangle  \\
\text{protected long}

The \langle cs \rangle should be a single control sequence, such as \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 \def, \edef, \gdef, and \xdef counterparts, but store and gstore will allow you to store macro parameters inside of them by using \unexpanded.

int \langle key \rangle = \langle cs \rangle  \\
\text{protected long}

The \langle cs \rangle should be a single control sequence, such as \foo. An int key will be a \TeX-count register. If \langle cs \rangle is defined yet, \newcount will be used to initialise it. The eint and xint versions will use \numexpr to allow basic computations in their \langle value \rangle. The gint and xint variants set the register globally.

dimen \langle key \rangle = \langle cs \rangle  \\
\text{protected long}

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

skip \langle key \rangle = \langle cs \rangle  \\
\text{protected long}

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

toks \langle key \rangle = \langle cs \rangle  \\
\text{protected long}

The \langle cs \rangle should be a single control sequence, such as \foo. Store \langle value \rangle inside of a toks-register. The g variants use \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 \newtoks.

box \langle key \rangle = \langle cs \rangle  \\
\text{protected long}

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

meta \langle key \rangle = \langle (key)=(value), \ldots \rangle  \\
\text{protected long}

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)=(value) \rangle list with \#1. It works by calling a sub-\ekvset on the \langle (key)=(value) \rangle list, so a set key will only affect that \langle (key)=(value) \rangle list and not the current \ekvset.

nmeta \langle key \rangle = \langle (key)=(value), \ldots \rangle  \\
\text{protected long}

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

smeta \langle key \rangle = \langle (set)\langle (key)=(value), \ldots \rangle \rangle  \\
\text{protected long}

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)=(value) \rangle list inside of \langle set \rangle, so is equal to \ekvset{(set)}{(key)=(value), \ldots}.
snmeta snmeta ⟨key⟩ = {(set)⟩{(key)=⟨value⟩}, ...} protected long

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⟩.

set set ⟨key⟩ = {(set)} protected long

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

choice choice ⟨key⟩ = {(⟨value⟩)=⟨definition⟩}, ...} protected long

Defines ⟨key⟩ to be a choice key, meaning it will only accept a limited set of values. You should define each possible ⟨value⟩ inside of the ⟨value⟩=⟨definition⟩ list. If a defined ⟨value⟩ is passed to ⟨key⟩ the ⟨definition⟩ will be left in the input stream. You can make individual values protected inside the ⟨value⟩=⟨definition⟩ list. By default a choice key is expandable, an undefined ⟨value⟩ will throw an error in an expandable way.

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
  int keyE = \keyE
  dimen keyF = \keyF
  skip keyG = \keyG
  toks keyH = \keyH
  default keyH = \empty text
  box keyI = \keyI
  qdefault keyI = text
  choice keyJ = 
  { 
    protected 1 = \texttt{a} 
    2 = b
    3 = c 
  }
\begin{verbatim}
,4 = d
,5 = e
}
,edefault  keyJ = 2
,meta     keyK = {keyA={#1},keyB=false}
,set      setB = B
}
\end{verbatim}

1.5 License

Copyright © 2020 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 \textsc{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 Generic Code

The rest of this implementation will be the generic code.

Load \textsc{expkv} if the package didn’t already do so – since \textsc{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 \textsc{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}
\ekvdVersion
\ekvdDate
\end{verbatim}

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

\begin{verbatim}
\def\ekvdVersion{0.1a}
\def\ekvdDate{2020-02-29}
\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 11 for now.

\begin{verbatim}
\expandafter\chardef\csname ekvd@tmp\endcsname=\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.

\begin{verbatim}
\ekvd@long \ekvd@prot \ekvd@clear@prefixes \ekvd@empty
\end{verbatim}

\texttt{\expkv@def} will use \texttt{\ekvd@long} and \texttt{\ekvd@prot} to store whether a key should be defined as \texttt{\long} or \texttt{\protected}, and we have to clear them for every new key. By default they’ll just be empty.

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


\let\ekvd@long\ekvd@empty
\let\ekvd@prot\ekvd@empty
}
\ekvd@clear@prefixes

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

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

\begin{verbatim}
\protected\def\ekvdefinekeys#1{\def\ekvd@set{#1}\ekvparse\ekvd@noarg\ekvd@}
\end{verbatim}

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

\ekvd@noarg
\ekvd@

\ekvd@noarg just places a special marker and gives control to \ekvd@. \ekvd@ 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.

\begin{verbatim}
\protected\def\ekvd@noarg#1{\ekvd@{#1}\ekvd@noarg@mark}
\protected\long\def\ekvd@#1#2{\ekvd@clear@prefixes\ekvd@ifspace{#1}{\ekvd@prefix\ekv@mark#1\ekv@stop{#2}}{\ekvd@err@missing@prefix{#1}}}
\end{verbatim}

(End definition for \texttt{\ekvd@noarg} and \texttt{\ekvd@}.)

\ekvd@prefix
\ekvd@prefix@

\exp@per 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},
\texttt{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).

\begin{verbatim}
\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}\ekv@undefined@prefix{#1}\@gobble}{}}
\end{verbatim}

(End definition for \texttt{\ekvd@prefix} and \texttt{\ekvd@prefix@}.)

8
The \texttt{\ekvd@prefix@after@p} type prefixes are all just modifying a following \texttt{\ekvd@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.

\begin{verbatim}
\protected\def\ekvd@prefix@after@p#1\% {\ekvd@ifspace{#1}{\ekvd@prefix#1\ekv@stop} {\expandafter\ekvd@err@missing@prefix\expandafter{\ekv@gobble@mark#1}\@gobble}}
\end{verbatim}

(End definition for \texttt{\ekvd@prefix@after@p}.)

\texttt{\ekvd@p@long} \texttt{\ekvd@p@protected} \texttt{\ekvd@p@protect}

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

\begin{verbatim}
\protected\def\ekvd@p@long{\let\ekvd@long\long\ekvd@prefix@after@p}
\protected\def\ekvd@p@protected{\let\ekvd@prot\protected\ekvd@prefix@after@p}
\let\ekvd@p@protect\ekvd@p@protected
\end{verbatim}

(End definition for \texttt{\ekvd@p@long}, \texttt{\ekvd@p@protected}, and \texttt{\ekvd@p@protect}.)

\subsection{Key Types}

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

\begin{verbatim}
\protected\def\ekvd@t@set#1#2\% {\ekvd@assert@not@long{set #1}\ekvd@assert@not@protected{set #1} {\ekvd@ifnoarg{#2} {\ekvdefNoVal\ekvd@set{#1}{\ekvchangeset{#1}}} {\ekv@ifempty{#2} {\ekvd@err@missing@definition{set #1}} {\ekvdefNoVal\ekvd@set{#1}{\ekvchangeset{#2}}}}}}
\end{verbatim}

(End definition for \texttt{\ekvd@t@set}.)

\texttt{\ekvd@type@noval} \texttt{\ekvd@t@noval} \texttt{\ekvd@t@enoval}

Another pretty simple type, \texttt{noval} just needs to assert that there is a definition and that \texttt{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 \texttt{@type@}. \texttt{noval} and \texttt{enoval} are so similar that we can use such a \texttt{@type@} macro, even if we could've done \texttt{noval} in a slightly faster way without it.

\begin{verbatim}
\protected\long\def\ekvd@type@noval#1#2#3#4\% {\ekvd@assert@arg{#1\#3}{#4} {\ekvd@assert@not@long{\#1\#3} {\ekvd@prot#2\ekvd@tmp{#4} {\ekvletNoVal\ekvd@set{\#3}\ekvd@tmp}}}{}
\end{verbatim}

(End definition for \texttt{\ekvd@type@noval}.)
code is simple as well, ecode has to use \edef on a temporary macro, since \expv doesn't provide an \ekvedef.

\protected\long\def\ekvd@type@code\ekvd@t@code\ekvd@t@ecode
{(End definition for \ekvd@type@code, \ekvd@t@code, and \ekvd@t@ecode.)}

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

\protected\long\def\ekvd@t@edefault\ekvd@t@edefault\ekvd@t@edefault\ekvd@t@edefault\ekvd@t@edefault\ekvd@t@edefault
{(End definition for \ekvd@type@edefault, \ekvd@t@edefault, and \ekvd@t@qdefault.)}
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 ⟨key⟩=true. The true and false actions are always just \let ting the macro in #7 to some other macro (e.g., \iftrue).

\protected\def\ekvd@type@bool#1#2#3#4#5#6#7{% 
  \ekvd@assert@filledarg{#1bool#2 #6}{#7}% 
  \ekvd@newlet#7#5% 
  \ekvd@type@choice{#1bool#2}{#6}% 
  \protected\expandafter\def
  \csname ekvd@choice@name\ekvd@set{#6}{true}\endcsname{#3\let#7#4}% 
  \protected\expandafter\def
  \csname ekvd@choice@name\ekvd@set{#6}{false}\endcsname{#3\let#7#5}% 
}% 
\protected\def\ekvd@t@bool{}{}{}\iftrue\iffalse
\protected\def\ekvd@t@gbool{}g{}\global\iftrue\iffalse
\protected\def\ekvd@t@boolTF{}{}{}\@firstoftwo\@secondoftwo
\protected\def\ekvd@t@gboolTF{g{}TF}\global\@firstoftwo\@secondoftwo

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

\begin{verbatim}
\protected\def\ekvd@type@box#1#2#3#4\%
{\ekvd@assert@filledarg{#1box #3}{#4}\%
 \ekvd@newreg#4{box}\%
 \protected\ekvd@long\ekvdef\ekvd@set{#3}\%
 (#2\setbox#4\hbox{\begingroup##1\endgroup})}%
}\%
\end{verbatim}

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

Similar to box, but set the toks.

\begin{verbatim}
\protected\def\ekvd@type@toks#1#2#3#4\%
{\ekvd@assert@filledarg{#1toks #3}{#4}\%
 \ekvd@newreg#4{toks}\%
 \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4\expandafter{\the#4##1}}%
}\%
\end{verbatim}

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

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

\begin{verbatim}
\protected\def\ekvd@type@apptoks#1#2#3#4\%
{\ekvd@assert@filledarg{#1apptoks #3}{#4}\%
 \ekvd@newreg#4{toks}\%
 \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4\expandafter{\the#4##1}}%
}\%
\end{verbatim}

(End definition for \ekvd@type@apptoks, \ekvd@t@apptoks, and \ekvd@t@gapptoks.)

The \ekvd@type@reg can handle all the types for which the assignment will just be \{register\}=(\value{}).

\begin{verbatim}
\protected\def\ekvd@type@reg#1#2#3#4#5#6#7\%
{\ekvd@assert@filledarg{#1 #6}{#7}\%
 \ekvd@newreg#7(#2)%
}\%
\end{verbatim}

(End definition for \ekvd@type@apptoks, \ekvd@t@apptoks, and \ekvd@t@gapptoks.)
\protected\ekvd@long\ekvdef\ekvd@set{#6}{#3\#7=#4\#1\#5}\relax\%
}\
\protected\def\ekvd@t@int{\ekvd@type@reg{int}{count}{}{}}
\protected\def\ekvd@t@eint{\ekvd@type@reg{eint}{count}\numexpr{}}
\protected\def\ekvd@t@gint{\ekvd@type@reg{gint}{count}\global{}}
\protected\def\ekvd@t@xint{\ekvd@type@reg{xint}{count}\global\numexpr{}}
\protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{dimen}{}{}}
\protected\def\ekvd@t@edimen{\ekvd@type@reg{edimen}{dimen}\dimexpr{}}
\protected\def\ekvd@t@gdimen{\ekvd@type@reg{gdimen}{dimen}\global{}}
\protected\def\ekvd@t@xdimen{\ekvd@type@reg{xdimen}{dimen}\global\dimexpr{}}
\protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{skip}{}{}}
\protected\def\ekvd@t@eskip{\ekvd@type@reg{eskip}{skip}\glueexpr{}}
\protected\def\ekvd@t@gskip{\ekvd@type@reg{gskip}{skip}\global{}}
\protected\def\ekvd@t@xskip{\ekvd@type@reg{xskip}{skip}\global\glueexpr{}}

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

\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{\ekvd@type@store{}\edef}
\protected\def\ekvd@type@store\ekvd@t@store\ekvd@t@gstore{\ekvd@type@store{g}\edef}

(End definition for \ekvd@type@store, \ekvd@t@store, and \ekvd@t@gstore.)

\ekvd@type@estore\ekvd@t@estore\ekvd@t@xstore

And the straight forward estore types.
\protected\def\ekvd@type@estore{\ekvd@type@estore{}\edef}
\protected\def\ekvd@type@estore\ekvd@t@estore\ekvd@t@xstore{\ekvd@type@estore{x}\edef}

(End definition for \ekvd@type@estore, \ekvd@t@estore, and \ekvd@t@xstore.)

\ekvd@type@meta\ekvd@type@meta\ekvd@t@meta\ekvd@t@meta

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{\ekvd@type@meta{}\edef}
\protected\def\ekvd@type@meta\ekvd@t@meta\ekvd@t@meta{\ekvd@type@meta\expandafter\expandafter\expandafter\expandafter\expandafter{\ekvd@type@meta}{3}{5}}

(End definition for \ekvd@type@meta, \ekvd@t@meta, and \ekvd@t@meta.)
The `ekvd@type@meta` type is similar to `meta`, but needs two arguments inside of ⟨value⟩, such that the first is the ⟨set⟩ for which the sub-`ekvset` and the second is the ⟨key⟩=⟨value⟩ list.

```latex
\protected\long\def\ekvd@type@meta\#1\#2\%
{\ekvd@prot\def\ekvd@tmp\#2{\ekvset\#1\{\#3\}}\%
\protected\def\ekvd@t@meta{\ekvd@type@meta{}\ekvlet{##1}}
\protected\long\def\ekvd@t@nmeta\#1\#2\%
{\ekvd@assert@not@long{nmeta \#1}\%\ekvd@type@meta \#1\#2\%}
(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 @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`.

```latex
\protected\def\ekvd@type@choice\#1\#2\%
{\ekvd@assert@not@long{\#1 \#2}\%\expandafter\ekvd@type@choice@\expandafter{\@secondoftwo\#2}{\#1}{\#2}\%
\protected\long\def\ekvd@t@choice{\ekvd@type@choice{}\ekvlet{##1}}
(End definition for `ekvd@type@meta` and others.)
```

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

```
\protected\long\def\ekvd@type@smeta\#1\#2\#3\#4\#5\%
{\ekvd@assert@twoargs{s\#1meta \#4}{\#5}\%
\expandafter\ekvd@type@meta\expandafter{\@secondoftwo\#5}{\#3}{\#4}{\#2}{\ekvd@set}{\#1}{\ekvd@tmp}
\%}
\protected\long\def\ekvd@type@snmeta\#1\#2\%
{\ekvd@assert@not@long{snmeta \#1}\%\ekvd@type@smeta \#1\#2\%}
(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 @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\#1\#2\%
{\ekvd@assert@not@long{\#1 \#2}\%\expandafter\ekvd@type@choice@\expandafter{\@secondoftwo\#2}{\#1}{\#2}\%
\protected\long\def\ekvd@t@choice{\ekvd@type@choice{}\ekvlet{##1}}
(End definition for `ekvd@type@meta` and others.)
```
\texttt{\texttt{\textbackslash}ekvd@populate@choice} just uses \texttt{\texttt{\textbackslash}ekvp\textbackslash}parse and then gives control to \texttt{\texttt{\textbackslash}ekvd@populate@choice@noarg}, which throws an error, and \texttt{\texttt{\textbackslash}ekvd@populate@choice@}. 

\begin{verbatim}
\protected\def\ekvd@populate@choice
\{\ekvp\texttt{\textbackslash}ekvd@populate@choice@noarg\ekvd@populate@choice@
\}
\protected\long\def\ekvd@populate@choice@noarg#1\%
\{\expandafter\ekvd@err@missing@definition\expandafter{\ekvd@set@choice : #1}\%
\}
\end{verbatim}

\texttt{\texttt{\textbackslash}ekvd@populate@choice@} runs the prefix-test, if there is none we can directly define the choice, for that \texttt{\texttt{\textbackslash}ekvd@set@choice} will expand to the current choice-key's name, which will have been defined by \texttt{\texttt{\textbackslash}ekvd@t@choice}. If there is a prefix run the prefix grabbing routine, which was altered for \texttt{\texttt{\textbackslash}type@choice}. 

\begin{verbatim}
\protected\long\def\ekvd@populate@choice@#1#2\%
\{\ekvd@clear@prefixes\expandafter\ekvd@assert@arg\expandafter{\ekvd@set@choice : #1}{#2}\%
\{\ekvd@ifspace{#1}\{\ekvd@choice@prefix\ekv@mark#1\ekv@stop\%
\{\expandafter\def\csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname{#2}\%
\}
\}
\protected\def\ekvd@choice@prefix#1
\{\ekv@strip{#1}\ekvd@choice@prefix@\ekv@mark\%
\protected\def\ekvd@choice@prefix@#1#2\ekv@stop\%
\{\ekv@ifdefined{ekvd@choice@p@#1}\{\csname ekvd@choice@p@#1\endcsname\%
\ekvd@ifspace{#1}\{\ekv@gobble@mark#1\ekv@stop\%
\{\ekvd@prot\expandafter\def\csname ekv@strip{#2}\{\ekvd@choice@name\ekvd@set\ekvd@set@choice\%
\endcsname\%
\}
\}
\%
\{\ekvd@err@undefined@prefix{#1}\@gobble\%
\}
\protected\def\ekvd@choice@p@protected{\let\ekvd@prot\protected}
\let\ekvd@choice@p@protect\ekvd@choice@p@protected
\protected\def\ekvd@choice@p@long\ekvd@ifspace#1\%
\{\expandafter\ekvd@choice@p@long@\expandafter{\ekv@gobble@mark#1}\%
\}
\end{verbatim}
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.

2.2.2 \textbf{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.

Tests

This macro serves as a flag for the case that no \texttt{value} was specified for a key. As such it is not a test, but exists only for some tests.
While we can reuse many of the internals of `expkv` the specific case for this branch wasn’t needed by `expkv` and hence isn’t defined. We’ll need it, so we define it.

```
\long\def\ekvd@fi@firstoftwo{\fi\@secondoftwo\#1\#2{\fi#1}
```

*(End definition for `ekvd@fi@firstoftwo`)*

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⟩`.

```
\protected\def\ekvd@newlet#1#2{%
  \unless\ifdefined#1\let#1#2\fi%
\protected\def\ekvd@newreg#1#2{%
  \unless\ifdefined#1\csname new#2\endcsname#1\fi%
```

*(End definition for `ekvd@newlet` and `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#2{%
  \ekvd@ifnottwoargs{#2}{\ekvd@err@missing@definition{#1}}%
\long\def\ekvd@ifnottwoargs#1{%
  \if\ekvd@ifempty@gtwo#1\ekv@ifempty@B\ekv@ifempty@false\ekv@ifempty@A\@firstoftwo
```

*(End definition for `ekvd@assert@twoargs`, `ekvd@ifnottwoargs`, and `ekvd@ifempty@gtwo`)*

The test for an argument is just an `\ifx` comparison with our `noarg@mark`.

```
\long\def\ekvd@assert@arg#1#2{%
  \ifx\ekvd@noarg@mark#1\
  \fi%
```

*(End definition for `ekvd@assert@arg` and `ekvd@ifnoarg`)*
Some key-types don't want to be `\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.

```latex
\long\def\ekvd@assert@not@long#1% {
\ifx\ekvd@long\long\ekvd@err@no@long{#1}\fi
}\long\def\ekvd@assert@not@protected#1% {
\ifx\ekvd@prot\protected\ekvd@err@no@protected{#1}\fi
}
```

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

```latex
\long\def\ekvd@ifspace#1% {
\ifx\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@A
}
```

### 2.2.4 Messages

Most messages of `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:

```latex
\protected\def\ekvd@err@missing@definition#1% {\errmessage{expkv-def Error: Missing definition for key `\unexpanded{#1}'}}
\protected\def\ekvd@err@missing@prefix#1% {\errmessage{expkv-def Error: Missing prefix for key `\unexpanded{#1}'}}
\protected\def\ekvd@err@undefined@prefix#1% {\errmessage{expkv-def Error: Undefined prefix for key `\unexpanded{#1}'}}
\protected\def\ekvd@err@undefined@key#1% {\errmessage{expkv-def Error: Undefined key for key `\unexpanded{#1}'}}
```
The expandable error messages use \texttt{\ekvd@err}, which is just like \texttt{\ekv@err} from \expv or the way expl3 throws expandable error messages. It uses an undefined control sequence to start the error message. \texttt{\ekvd@err@choice@invalid} will have to use this mechanism to throw its message. Also we have to retrieve the name parts of the choice in an easy way, so we use parentheses of catcode 8 here, which should suffice in most cases to allow for a correct separation.

\begin{verbatim}
\def\ekvd@err@choice@invalid#1\{\ekvd@choice@name#1\}\ekv@stop
\def\ekvd@choice@name#1#2#3\{ekvd#1(#2)#3\}
\def\ekvd@err{invalid choice '#3' ('#2', set '#1')}\}
\end{verbatim}

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

<table>
<thead>
<tr>
<th>A</th>
<th>int</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>apptoks</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>long</td>
<td>3</td>
</tr>
<tr>
<td>bool</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>boolTF</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>box</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>meta</td>
<td>4</td>
</tr>
<tr>
<td>choice</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>code</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>nmeta</td>
<td>4</td>
</tr>
<tr>
<td>default</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>dimen</td>
<td>noval</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>protected</td>
<td>3</td>
</tr>
<tr>
<td>ecode</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>edefault</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>edimen</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>eint</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>\ekvchangeset</td>
<td></td>
<td>68, 72</td>
</tr>
<tr>
<td>\ekvDate</td>
<td></td>
<td>2, 5, 9, 15</td>
</tr>
<tr>
<td>\ekvdef</td>
<td></td>
<td>167, 178, 188, 198, 218, 228</td>
</tr>
<tr>
<td>\ekvdefinekeys</td>
<td></td>
<td>2, 27</td>
</tr>
<tr>
<td>\ekvdefNoVal</td>
<td></td>
<td>68, 72, 148</td>
</tr>
<tr>
<td>\ekvVersion</td>
<td></td>
<td>2, 5, 9, 15</td>
</tr>
<tr>
<td>\ekvifdefined</td>
<td></td>
<td>100, 119, 133</td>
</tr>
<tr>
<td>\ekylet</td>
<td></td>
<td>91, 246, 264, 277</td>
</tr>
<tr>
<td>\ekyletNoVal</td>
<td></td>
<td>81, 108, 124, 250, 268</td>
</tr>
<tr>
<td>\ekvparse</td>
<td></td>
<td>30, 281</td>
</tr>
<tr>
<td>\ekvset</td>
<td></td>
<td>244</td>
</tr>
<tr>
<td>enoval</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>eskip</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>estore</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>qdefault</td>
<td>3</td>
</tr>
<tr>
<td>gapptoks</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>gbool</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>gboolTF</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>gbox</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>gdimen</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>gint</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>gskip</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>gstore</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>gtoks</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>initial</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

**\TeX and \LaTeX commands:**

- `\@firstofone` 442, 458
- `\@firstoftwo` 159, 161, 262, 374, 397, 411
- `\@gobble` 48, 57, 319, 352
- `\@secondoftwo` 159, 161, 256, 357, 387
- `\ekv@err@` 458
- `\ekv@gobble@mark` 56, 325
- `\ekv@ifdefined` 43, 46, 307
- `\ekv@ifempty` 70, 398
- `\ekv@ifempty@` 376, 415
- `\ekv@ifempty@A` 374, 376, 411, 415
- `\ekv@ifempty@B` 373, 374, 410, 411
- `\ekv@ifempty@false` 374, 411
- `\ekv@mark` 37, 40, 293, 303
- `\ekv@name` 106, 123, 137
- `\ekv@stop` 37, 41, 54, 293, 305, 311, 437, 447, 460
- `\ekv@strip` 40, 44, 303, 315
- `\ekv@` 30, 32
- `\ekv@assert@arg` 144, 218, 268, 277, 357, 387
- `\ekv@assert@filledarg` 144, 164, 175, 185, 195, 215, 225, 235, 389
\ekvd@t@xdimen \hspace{1pt} 193
\ekvd@t@xint \hspace{1pt} 193
\ekvd@t@xskip \hspace{1pt} 193
\ekvd@t@xstore \hspace{1pt} 223
\ekvd@t@tmp \hspace{1pt} 2, 80, 81, 90, 91, 103, 108, 122, 124, 237, 238, 239, 244, 257, 273, 277, 278, 464
\ekvd@t@type@apptoks \hspace{1pt} 183
\ekvd@t@type@bool \hspace{1pt} 142
\ekvd@t@type@box \hspace{1pt} 162
\ekvd@t@type@choice \hspace{1pt} 147, 270
\ekvd@t@type@code \hspace{1pt} 86
\ekvd@t@type@default \hspace{1pt} 96
\ekvd@t@type@estore \hspace{1pt} 223
\ekvd@t@type@meta \hspace{1pt} 233
\ekvd@t@type@meta@ \hspace{1pt} 233, 262
\ekvd@t@type@noval \hspace{1pt} 75
\ekvd@t@type@reg \hspace{1pt} 193
\ekvd@t@type@reg@ \hspace{1pt} 193
\ekvd@t@type@smeta \hspace{1pt} 252
\ekvd@t@type@smeta@ \hspace{1pt} 252
\ekvd@t@type@store \hspace{1pt} 213
\ekvd@t@type@toks \hspace{1pt} 173
toks \hspace{1pt} 4

xdimen \hspace{1pt} 4
xint \hspace{1pt} 4
xskip \hspace{1pt} 4
xstore \hspace{1pt} 4