Abstract

\texttt{expkvedf} provides a small \langle \text{key} \rangle = \langle \text{value} \rangle interface to define keys for \texttt{expkve}. Key-types are declared using prefixes, similar to static typed languages. The stylised name is \texttt{expkvedf} 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
   1.1 Macros ......................................................... 2
   1.2 Prefixes ..................................................... 2
      1.2.1 p-Prefixes ............................................. 2
      1.2.2 t-Prefixes ............................................. 3
   1.3 Bugs ......................................................... 6
   1.4 Example ...................................................... 6
   1.5 License ....................................................... 7

2 Implementation
   2.1 The \LaTeX{} Package ........................................ 8
   2.2 The Generic Code .......................................... 8
      2.2.1 Key Types ............................................. 10
      2.2.2 Key Type Helpers ................................... 18
      2.2.3 Tests .................................................. 18
      2.2.4 Messages ............................................. 20

Index

\*jspratte@yahoo.de
1 Documentation

Since the trend for the last couple of years goes to defining keys for a \langle key \rangle=\langle value \rangle interface using a \langle key \rangle=\langle 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 key \rangle=\langle value \rangle interfaces with \langle key \rangle=\langle value \rangle interfaces.

Unlike many of the other established \langle key \rangle=\langle 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: p-type, which are equivalent to \TeX\'s prefixes like \texttt{\long}, and t-type defining the type of the key. For a description of the available p-prefixes take a look at \texttt{subsubsection 1.2.1}, the 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\def} in the same mode as well. To use it, just use one of

\begin{verbatim}
\usepackage{expkv-def} % \LaTeX
\input expkv-def % plain \TeX
\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 p-type prefixes followed by one t-type prefix. The syntax of \langle value \rangle 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 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 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.
| **protected** | The following key will be defined \protected. Note that key-types which can't be defined expandable will always use \protected. |
| **long** | The following key will be defined \long. |

### 1.2.2 t-Prefixes

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

| code | code \{key\} = \{definition\} protected long |
| ecode | Define \{key\} to expand to \{definition\}. The \{key\} will require a \{value\} for which you can use \#1 inside \{definition\}. The ecode variant will fully expand \{definition\} inside an \edef. |
| noval | noval \{key\} = \{definition\} protected long |
| enoval | The noval type defines \{key\} to expand to \{definition\}. The \{key\} will not take a \{value\}. enoval fully expands \{definition\} inside an \edef. |
| default | default \{key\} = \{definition\} protected long |
| qdefault | This serves to place a default \{value\} for a \{key\} that takes an argument, the \{key\} can be of any argument-grabbing kind, and when used without a \{value\} it will be passed \{definition\} instead. The qdefault variant will expand the \{key\}'s code once, so will be slightly quicker, but not change if you redefine \{key\}. The edefault on the other hand fully expands the \{key\}-code with \{definition\} as its argument inside of an \edef. |
| edefault | |
| initial | initial \{key\} = \{value\} protected long |
| oinitial | With initial you can set an initial \{value\} for an already defined argument taking \{key\}. It'll just call the key-macro of \{key\} and pass it \{value\}. The oinitial variant will expand \{value\} using an \edef expansion prior to passing it to the key-macro and the oinitial variant will expand the first token in \{value\} once. |
bool \langle key \rangle = \langle cs \rangle

The \langle cs \rangle should be a single control sequence, such as \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 \TeX-ifs so either be \iftrue or \iffalse. The boolTF and gboolTF variants will both take two arguments and if true the first will be used else the second, so they are always either \@firstoftwo or \@secondoftwo. The variants with a leading g will set the control sequence globally, the others locally. If \langle cs \rangle is not yet defined it'll be initialised as the false version. Note that the initialisation is not done with \newif, so you will not be able to do \iftrue outside of the \langle key \rangle=\langle value \rangle interface, but you could use \newif yourself. Even if the \langle key \rangle will not be \protected 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 \edef \langle key \rangle = \false without an issue to change the default behaviour to execute the \false choice).

store \langle key \rangle = \langle cs \rangle

The \langle cs \rangle should be a single control sequence, such as \iffoo. 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.

data \langle key \rangle = \langle cs \rangle

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

dataT \langle key \rangle = \langle cs \rangle

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

int \langle key \rangle = \langle cs \rangle

The \langle cs \rangle should be a single control sequence, such as \iffoo. An int key will be a TeX-count register. If \langle cs \rangle isn't 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

The \langle cs \rangle should be a single control sequence, such as \iffoo. This is just like int but uses a dimen register, \newdimen and \dimexpr instead.
skip \(\text{skip } \langle \text{key} \rangle = \langle cs \rangle\) \hfill \text{protected long}

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

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

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

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

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

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

nmeta \(\text{nmeta } \langle \text{key} \rangle = \langle(key)=(value), \ldots\rangle\) \hfill \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 \(\langle key \rangle=(value)\) list is static.

smeta \(\text{smeta } \langle \text{key} \rangle = \langle(set)=(key)=(value), \ldots\rangle\) \hfill \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 \(\langle key \rangle=(value)\) list inside of \(\langle set \rangle\), so is equal to \texttt{\textbackslash ekvset\langle(set)\rangle\langle(key)=(value), \ldots\rangle}.

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

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

set \(\text{set } \langle \text{key} \rangle = \langle(set)\rangle\) \hfill \text{protected long}

This will define \(\langle key \rangle\) to change the set of the current \texttt{\textbackslash 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 set key will not take a \(\langle value \rangle\). Note that just like in expr 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 \(\text{choice } \langle \text{key} \rangle = \langle(value)=(definition), \ldots\rangle\) \hfill \text{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 \(\langle value \rangle=(definition)\) 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 \(\langle value \rangle=(definition)\) list. By default a choice key is expandable, an undefined \(\langle value \rangle\) 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
, data keyE = \keyE
, dataT keyF = \keyF
, int keyG = \keyG
, dimen keyH = \keyH
, skip keyI = \keyI
, toks keyJ = \keyJ
, default keyJ = \empty t e s t
, box keyK = \keyK
, qdefault keyK = text
, choice keyL =
{ }
, protected 1 = \texttt{a}
, 2 = b
, 3 = c
, 4 = d
, 5 = e
}
, edefault keyL = 2
, meta keyM = {keyA=#1, keyB=false}
, data
}
\newcommand\Person[1]{
\begingroup
%

A person requires a name
A person requires an age
A person requires a name
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 Generic Code

The rest of this implementation will be the generic code.

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

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

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

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

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

\begin{verbatim}
\def\ekvdVersion{0.2a}
\def\ekvdDate{2020-04-04}
\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.

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'\@
\\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{}
\end{verbatim}

\texttt{\protected} will use \texttt{\ekvd@clear@prefixes}
\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 (key)=(value) list, the interpretation will be done by \ekvd@noarg and \ekvd@. The (set) for which the keys should be defined is stored in \ekvd@set.

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

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

\ekvd@noarg

\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.

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

(End definition for \ekvd@noarg and \ekvd@)

\ekvd@prefix

\ekvd@prefix separates prefixes into two groups, the first being prefixes in the \TeX{} sense (long and protected) which use @p@ in their name, the other being key-types (@c@, @i@, etc.) which use @t@ instead. \ekvd@prefix splits at the first space and checks whether its a @p@ or @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@{#1}\ekv@mark}
\protected\def\ekvd@prefix@#1#2{\ekv@ifdefined{ekvd@t@#1}{\csname ekvd@t@#1\endcsname{#2}}{\ekvd@err@undefined@prefix{#1}\@gobble}}

(End definition for \ekvd@prefix and \ekvd@prefix@)
The @p@ type prefixes are all just modifying a following @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.

```latex
\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 definition for \ekvd@prefix@after@p)

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

```latex
\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 definition for \ekvd@p@long, \ekvd@p@protected, and \ekvd@p@protect.)

### 2.2.1 Key Types

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

```latex
\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 definition for \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.

```latex
\protected\long\def\ekvd@type@noval#1% 
  {\ekvd@assert@arg{#1noval #3}{#4}%
    \ekvd@assert@not@long{#1noval #3}%
    \ekvd@prot#2\ekvd@tmp{#4}%
    \ekvletNoVal\ekvd@set{#3}\ekvd@tmp
    {#4}}
```
code is simple as well, ecode has to use \edef on a temporary macro, since expkv doesn't provide an \ekvedef.

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

default is too different from default and qdefault to reuse the @type@ macro, as it doesn't need \unexpanded inside of \edef.
\ekvifdefined\ekvd@set{#1}%
  {%
    \ekvd@assert\@not@long{edefault #1}%
    \ekvd@prot\edef\ekvd@tmp
    {\csname\ekv@name\ekvd@set{#1}\endcsname{#2}}%
    \ekvletNoVal\ekvd@set{#1}\ekvd@tmp
  }%
  {\ekvd@err@undefined@key{#1}}%
}%

(End definition for \ekvd@t@edefault.)

\ekvd@t@initial
\ekvd@t@oinitial
\ekvd@t@einitial
\long\def\ekvd@t@initial#1#2%
  {%
    \ekvd@assert@arg{initial #1}{#2}%
    {%
      \ekvifdefined\ekvd@set{#1}%
      {%\ekvd@assert\@not@long{initial #1}%
        \ekvd@assert\@not@protected{initial #1}%
        {\csname\ekv@name\ekvd@set{#1}\endcsname{#2}}%
      }%
      {\ekvd@err@undefined@key{#1}}%
    }%
  }

\long\def\ekvd@t@oinitial#1#2%
  {%\ekvd@assert@arg{oinitial #1}{#2}%
    {%\ekvifdefined\ekvd@set{#1}%
      {%\ekvd@assert\@not@long{oinitial #1}%
        \ekvd@assert\@not@protected{oinitial #1}%
        {\csname\ekv@name\ekvd@set{#1}\expandafter\endcsname\expandafter{#2}}%
      }%
      {\ekvd@err@undefined@key{#1}}%
    }%
  }

\long\def\ekvd@t@einitial#1#2%
  {%\ekvd@assert@arg{einitial #1}{#2}%
    {%\ekvifdefined\ekvd@set{#1}%
      {%\ekvd@assert\@not@long{einitial #1}%
        \ekvd@assert\@not@protected{einitial #1}%
        \edef\ekvd@tmp{#2}%
        {\csname\ekv@name\ekvd@set{#1}\expandafter\endcsname\expandafter{\ekvd@tmp}}%
      }%
      {\ekvd@err@undefined@key{#1}}%
    }%
  }
The boolean types are a quicker version of a choice that accept true and false, and set up the NoVal action to be identical to \langle key\rangle=true. The true and false actions are always just \letting 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 \ncname\ekvd@choice@name\ekvd@set{#6}{true}\endcsname \{#3\let#7#4\}\% 
  \protected\expandafter\def \ncname\ekvd@choice@name\ekvd@set{#6}{false}\endcsname \{#3\let#7#5\}\% 
}%% 
\protected\def\ekvd@t@bool{\ekvd@type@bool{}{}{}\iftrue\iffalse}\protected\def\ekvd@t@gbool{\ekvd@type@bool g{}\global\iftrue\iffalse}\protected\def\ekvd@t@boolTF{\ekvd@type@bool{}{TF}{}\@firstoftwo\@secondoftwo}\protected\def\ekvd@t@gboolTF{\ekvd@type@bool g{TF}\global\@firstoftwo\@secondoftwo}(End definition for \ekvd@type@bool and others.)

\protected\def\ekvd@type@data#1#2#3#4#5#6#7{% 
  \ekvd@assert@filledarg{#1data#2 #6}{#7}%% 
  \ekvd@newlet#7#3% 
  \protected\ekvd@long\ekvdef\ekvd@set{#6}{\long#4#7####1#5{####1{##1}}}%% 
}%% 
\protected\def\ekvd@t@data{\ekvd@type@data{}{}\@secondoftwo\def{####2}}\protected\def\ekvd@t@edata{\ekvd@type@data e{}\@secondoftwo\edef{####2}}\protected\def\ekvd@t@gdata{\ekvd@type@data g{}\@secondoftwo\gdef{####2}}\protected\def\ekvd@t@xdata{\ekvd@type@data x{}\@secondoftwo\xdef{####2}}\protected\def\ekvd@t@dataT{\ekvd@type@data{}T\@gobble\def{}}\protected\def\ekvd@t@edataT{\ekvd@type@data eT\@gobble\edef{}}\protected\def\ekvd@t@gdataT{\ekvd@type@data gT\@gobble\gdef{}}\protected\def\ekvd@t@xdataT{\ekvd@type@data xT\@gobble\xdef{}}(End definition for \ekvd@type@data and others.)

\ekvd@type@box Set up our boxes. Though we’re a generic package we want to be colour safe, so we put an additional grouping level inside the box contents, for the case that someone uses color.

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

\protected\def\ekvd@type@box#1#2#3#4{\ekvd@newreg\{#1#2#3#4\}#1}}}  
{}}}
\ekvd@assert@filledarg{#1 box #3}{#4}\
\ekvd@newreg#4{box}\
\protected\ekvd@long\ekvdef\ekvd@set{#3}{#2 setbox #4 hbox (begingroup #1 endgroup)}\
\ekvd@t@box\
\ekvd@t@toks\
\ekvd@t@gtoks\
\ekvd@type@reg\
\ekvd@t@int\
\ekvd@t@eint\
\ekvd@t@gint\
\ekvd@t@xint\
\ekvd@t@dimen\
\ekvd@t@edimen\
\ekvd@t@gdimen\
\ekvd@t@xdimen\
\ekvd@t@skip\
\ekvd@t@eskip\
\ekvd@t@gskip\
\ekvd@t@xskip\
\ekvd@assert@filledarg{#1 toks #3}{#4}\
\ekvd@newreg#4{toks}\
\protected\ekvd@long\ekvdef\ekvd@set{#3}{#2 setbox #4 \expandafter{\the#4 ##1}}\
\ekvd@t@apptoks\
\ekvd@t@apptoks\
\ekvd@t@gapptoks\
\ekvd@type@apptoks\
\ekvd@t@apptoks\
\ekvd@t@gapptoks\
\ekvd@type@apptoks\
\ekvd@t@apptoks\
\ekvd@t@gapptoks\
\ekvd@type@apptoks\
\ekvd@t@apptoks\
\ekvd@t@gapptoks\
\end{definition}
The none-expanding store types use an `\edef` or `\xdef` and `\unexpanded` to be able to also store # easily.

The straight forward `estore` types.

meta sets up things such that another instance of `\ekvset` will be run on the argument, with the same `{set}`.
\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 n\ekvletNoVal{(#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 \texttt{p\@\texttt{type prefixes as well (but \texttt{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}\protect\edef\ekvd@tmp##1{%\unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#2}{##1}}\}\ekvlet\ekvd@set{#2}\ekvd@tmp\}

\ekvd@populate@choice just uses \texttt{ekvpars} and then gives control to \texttt{ekvd@populate@choice@noarg}, which throws an error, and \texttt{ekvd@populate@choice@noarg}.
\expandafter\ekvd@err@missing@definition\expandafter{\ekvd@set@choice : #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@t@choice. If there is a prefix run the prefix grabbing
routine, which was altered for \@type@choice.
\protected\long\def\ekvd@populate@choice@{#1#2%\
\ekvd@clear@prefixes\
\expandafter\ekvd@assert@arg\expandafter{\ekvd@set@choice : #1}{#2}\
\ekv@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@stop{\
\expandafter\ekvd@assert\expandafter{
\ekvd@set@choice : long #1}\
}\protected\def\ekvd@choice@prefix@long\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname\n{#2}\
\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@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname\n{#2}\
\expandafter\ekvd@err\expandafter{\ekvd@set@choice : long #1}\
}
Finally we’re able to set up the `@t@choice` macro, which has to store the current choice-key’s name, define the key, and parse the available choices.

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

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

### 2.2.2 Key Type Helpers

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

\ekvd@h@choice
\ekvd@h@choice@  
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.

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

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

### 2.2.3 Tests

\ekvd@noarg@mark  
This macro serves as a flag for the case that no `<value>` was specified for a key. As such it is not a test, but exists only for some tests.

\def\ekvd@noarg@mark{\ekvd@noarg@mark}\ekvd@noarg@mark\%  
(End definition for `\ekvd@noarg@mark`.)

\ekvd@fi@firstoftwo  
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⟩.

\begin{verbatim}
\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{verbatim}

(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.

\begin{verbatim}
\long\def\ekvd@assert@twoargs#1#2{\ekvd@ifnottwoargs{#2}{\ekvd@err@missing@definition{#1}}}
\end{verbatim}

(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.

\begin{verbatim}
\long\def\ekvd@assert@arg#1#2{\ekvd@ifnoarg{#2}{\ekvd@err@missing@definition{#1}}}
\end{verbatim}

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

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

\begin{verbatim}
\long\def\ekvd@assert@filledarg#1#2{\ekvd@ifnoarg@or@empty{#2}{\ekvd@err@missing@definition{#1}}}
\end{verbatim}

(End definition for \ekvd@assert@filledarg and \ekvd@ifnoarg@or@empty.)
Some key-types don’t want to be \texttt{\textbackslash long} or \texttt{\textbackslash 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}
\long\def\ekvd@assert@not@long#1\%
\{
  \ifx\ekvd@long\long\ekvd@err@no@long{#1}\fi
\}
\end{verbatim}

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

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

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

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

### Messages

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

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

\begin{verbatim}
\protected\def\ekvd@err@missing@definition\%
\{
\errmessage{\expkv@def Error: Missing definition for key \texttt{\unexpanded{\textbackslash \textbackslash}}}\}
\}
\end{verbatim}

\begin{verbatim}
\protected\def\ekvd@err@missing@prefix\%
\{
\errmessage{\expkv@def Error: Missing prefix for key \texttt{\unexpanded{\textbackslash}}}\}
\}
\end{verbatim}

\begin{verbatim}
\protected\def\ekvd@err@undefined@prefix\%
\{
\errmessage{\expkv@def Error: Undefined prefix \texttt{\unexpanded{\textbackslash}}}\}
\}
\end{verbatim}

\begin{verbatim}
\protected\def\ekvd@err@undefined@key\%
\{
\errmessage{\expkv@def Error: Undefined key \texttt{\unexpanded{\textbackslash}}}\}
\}
\end{verbatim}

\begin{verbatim}
\protected\def\ekvd@err@protected\%
\{
\errmessage{\expkv@def Error: prefix 'protected' not accepted for \texttt{\unexpanded{\textbackslash}}}\}
\}
\end{verbatim}
The expandable error messages use \ekvd@err, which is just like \ekv@err from expl3 or the way expl3 throws expandable error messages. It uses an undefined control sequence to start the error message. \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.

\def\ekvd@err@choice@invalid#1\ekv@stop
\begingroup
\catcode40=8
\catcode41=8
\@firstofone{\endgroup
\def\ekvd@choice@name#1#2#3\ekv@stop
{\ekvd@err{invalid choice '#3' ('#2', set '#1')}\ekv@stop}
\begingroup
\edef\ekvd@err
\unexpanded{\long\def\ekvd@err\##1\ekv@stop}
\endgroup
(End definition for \ekvd@err@choice@invalid and others.)

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

\catcode’@=\ekvd@tmp
(End definition for \ekvd@err@choice@invalid and others.)
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>\gdef \ekvchangeset \ekvdDate \ekvset \ekvparse \ekvifdef \ekvdef \ekvversion \definekeys \letNoVal</th>
<th>200, 204</th>
</tr>
</thead>
<tbody>
<tr>
<td>apptoks</td>
<td>\edef \dimen \int \skip \store \toks</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>\edef \gdimen \gint \gskip \gstore \gtoks</td>
<td>5</td>
</tr>
<tr>
<td>bool</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>boolTF</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>box</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>choice</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>code</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>data</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>dataT</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>default</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>dimen</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>ecode</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>edata</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>edataT</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>edefault</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>einitial</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>eint</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvchangeset</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvdDate</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvdef ... 195, 211, 222, 232, 242, 262, 272</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvdefinekeys</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvdefNoVal</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvVersion</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvifdef</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvifdefined</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvlet</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvletNoVal</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvparse</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekvset</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>enoval</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>eskip</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>estore</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>gapptoks</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>gbool</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>gboolTF</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>gbox</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>gdata</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>gdataT</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>H</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>I</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>initial</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>int</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>L</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>long</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>Meta</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>M</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>nmeta</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>noval</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>ninitial</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>O</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>original</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>P</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>protect</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>protected</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>Q</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>qdefault</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>R</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>S</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>set</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>skip</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>smeta</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>snmeta</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>T</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\TeX\ and \LaTeX 2ε commands:</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekv@err@</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekv@gobble@mark</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekv@ifdefined</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekv@ifempty</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekv@ifempty@</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekv@ifempty@false</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekv@ifempty@falseA</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekv@ifempty@falseB</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
<tr>
<td>\ekv@mark</td>
<td>\edef \gboolTF \nmeta \noval \oninitial</td>
<td>3</td>
</tr>
</tbody>
</table>

22