expkv|def

a key-defining frontend for expkv

Jonathan P. Spratte*

2020-04-29 v0.3

Abstract

expkv|def provides a small ⟨*key*⟩=⟨*value*⟩ interface to define keys for expkv. Key-types are declared using prefixes, similar to static typed languages. The stylised name is expkv|def but the files use `expkv-def`, this is due to CTAN-rules which don't allow | in package names since that is the pipe symbol in *nix shells.

Contents

*jspratte@yahoo.de

1 Documentation

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

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

expkv|def is usable as generic code and as a LATEX package. It'll automatically load expkv in the same mode as well. To use it, just use one of

```
\usepackage{expkv-def} % LaTeX
\input expkv-def       % plainTeX
```

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.

`\ekvdefinekeys` `\ekvdefinekeys{⟨set⟩}{⟨key⟩=⟨value⟩, ...}`

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.

`\ekvdDate` `\ekvdVersion` 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 ⟨*key*⟩ at use-time, so omitting `long` doesn't mean that a ⟨*definition*⟩ can't contain a `\par` token, only that the ⟨*key*⟩ will not accept a `\par` in ⟨*value*⟩.

protected
protect

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
ecode

code ⟨*key*⟩ = {⟨*definition*⟩} protected long

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
enoval

noval ⟨*key*⟩ = {⟨*definition*⟩} protected long

The noval type defines ⟨*key*⟩ to expand to ⟨*definition*⟩. The ⟨*key*⟩ will not take a ⟨*value*⟩. enoval fully expands ⟨*definition*⟩ inside an \edef.

default
qdefault
edefault

default ⟨*key*⟩ = {⟨*definition*⟩} protected long

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.

initial
oinitial
einitial

initial ⟨*key*⟩ = {⟨*value*⟩} protected long

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 einitial 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
gbool
boolTF
gboolTF

`bool` ⟨*key*⟩ `=` ⟨*cs*⟩ protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\iffoo`. This will define ⟨*key*⟩ 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 ⟨*value*⟩ it'll have the same effect as if you use ⟨*key*⟩`=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 ⟨*cs*⟩ 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 `\footrue` outside of the ⟨*key*⟩=⟨*value*⟩ interface, but you could use `\newif` yourself. Even if the ⟨*key*⟩ will not be `\protected` 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 `edefault` ⟨*key*⟩ `= false` without an issue to change the default behaviour to execute the `false` choice).

store
estore
gstore
xstore

`store` ⟨*key*⟩ `=` ⟨*cs*⟩ protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. This will define ⟨*key*⟩ to store ⟨*value*⟩ inside of the control sequence. If ⟨*cs*⟩ 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
edata
gdata
xdata

`data` ⟨*key*⟩ `=` ⟨*cs*⟩ protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. This will define ⟨*key*⟩ to store ⟨*value*⟩ inside of the control sequence. But unlike the `store` type, the macro ⟨*cs*⟩ will be a switch at the same time, it'll take two arguments and if ⟨*key*⟩ was used expands to the first argument followed by ⟨*value*⟩ in braces, if ⟨*key*⟩ was not used ⟨*cs*⟩ 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 ⟨*value*⟩, the `gdata` and `xdata` variants will store ⟨*value*⟩ inside ⟨*cs*⟩ globally. The `p`-prefixes will only affect the key-macro, ⟨*cs*⟩ will always be expandable and `\long`.

dataT
edataT
gdataT
xdataT

`dataT` ⟨*key*⟩ `=` ⟨*cs*⟩ protected long

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

int
eint
gint
xint

`int` ⟨*key*⟩ `=` ⟨*cs*⟩ protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. An `int` key will be a TEX-count register. If ⟨*cs*⟩ 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 ⟨*value*⟩. The `gint` and `xint` variants set the register globally.

dimen
edimen
gdimen
xdimen

`dimen` ⟨*key*⟩ `=` ⟨*cs*⟩ protected long

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

skip
eskip
gskip
xskip

`skip` ⟨*key*⟩ `=` ⟨*cs*⟩ — `protected long`

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

toks
gtoks
apptoks
gapptoks

`toks` ⟨*key*⟩ `=` ⟨*cs*⟩ — `protected long`

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

box
gbox

`box` ⟨*key*⟩ `=` ⟨*cs*⟩ — `protected long`

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. Typesets ⟨*value*⟩ into a `\hbox` and stores the result in a box register. The boxes are colour safe. expkv|def doesn't provide a `vbox` type.

meta

`meta` ⟨*key*⟩ `= {`⟨*key*⟩`=`⟨*value*⟩`, ...}` — `protected long`

This key type can set other keys, 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 `\ekvsneak` using keys or similar macros in the way you normally could.

nmeta

`nmeta` ⟨*key*⟩ `= {`⟨*key*⟩`=`⟨*value*⟩`, ...}` — `protected long`

This key type can set other keys, the difference to `meta` is, that this key doesn't take a value, so the ⟨*key*⟩=⟨*value*⟩ list is static.

smeta

`smeta` ⟨*key*⟩ `= {`⟨*set*⟩`}{`⟨*key*⟩`=`⟨*value*⟩`, ...}` — `protected long`

Yet another `meta` variant. An `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*⟩`}{`⟨*key*⟩`=`⟨*value*⟩`, ...}`.

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
    ,data      keyE = \keyE
    ,dataT     keyF = \keyF
    ,int       keyG = \keyG
    ,dimen     keyH = \keyH
    ,skip      keyI = \keyI
    ,toks      keyJ = \keyJ
    ,default   keyJ = \empty test
    ,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
  }
```

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}}
      \Phobby{\item[Hobbies]}
    \end{description}
    \endgroup
  }
\Person{name=Jonathan P. Spratte, age=young, hobby=\TeX\ coding}
\Person{name=Some User, age=unknown, hobby=Reading Documentation}
\Person{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

Age *young*

Hobbies TEX coding

Some User

Age *unknown*

Hobbies Reading Documentation

Anybody

Age *any*

1.5 License

2 Implementation

2.1 The LaTeX Package

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

```
\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]
```

2.2 The Generic Code

The rest of this implementation will be the generic code.

Load expkv if the package didn't already do so – since 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 expkv to save us from retyping them.

```
\input expkv
```

We make sure that `expkv-def.tex` is only input once:

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

`\ekvdVersion` `\ekvdDate`

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

```
\def\ekvdVersion{0.3}
\def\ekvdDate{2020-04-29}
```

(*End definition for* `\ekvdVersion` *and* `\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 `\ekvd@tmp` to use `\ProvidesFile`, else this will expand to a `\relax` and do no harm.

```
\csname ekvd@tmp\endcsname
```

Store the category code of `@` to later be able to reset it and change it to 11 for now.

```
\expandafter\chardef\csname ekvd@tmp\endcsname=\catcode`\@
\catcode`\@=11
```

`\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 expkv|def was initialized.

`\ekvd@long` `\ekvd@prot` `\ekvd@clear@prefixes` `\ekvd@empty`

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

```
\def\ekvd@empty{}
\protected\def\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@` `\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\ekv@mark#1\ekv@stop{#2}}%
      {\ekvd@err@missing@prefix{#1}}%
  }
```

(*End definition for* `\ekvd@noarg` *and* `\ekvd@`.)

`\ekvd@prefix` `\ekvd@prefix@` expkv|def 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 (`code`, `int`, *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@\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{#2}}%
          {\ekvd@err@undefined@prefix{#1}\@gobble}%
      }%
  }
```

(*End definition for* `\ekvd@prefix` *and* `\ekvd@prefix@`.)

\ekvd@prefix@after@p 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.

```
\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`.)

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

```
\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

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

```
\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`.)

\ekvd@type@noval \ekvd@t@noval \ekvd@t@enoval Another pretty simple type, noval just needs to assert that there is a definition and that long wasn't specified. There are types where the difference in the variants is so small, that we define a common handler for them, those common handlers are named with @type@. noval and enoval are so similar that we can use such a @type@ macro, even if we could've done noval in a slightly faster way without it.

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

```
  }
\protected\def\ekvd@t@noval{\ekvd@type@noval{}\def}
\protected\def\ekvd@t@enoval{\ekvd@type@noval e\edef}
```

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

`\ekvd@type@code` `\ekvd@t@code` `\ekvd@t@ecode`

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

```
\protected\long\def\ekvd@type@code#1#2#3#4%
  {%
    \ekvd@assert@arg{#1code #3}{#4}
      {%
        \ekvd@prot\ekvd@long#2\ekvd@tmp##1{#4}%
        \ekvlet\ekvd@set{#3}\ekvd@tmp
      }%
  }
\protected\def\ekvd@t@code{\ekvd@type@code{}\def}
\protected\def\ekvd@t@ecode{\ekvd@type@code e\edef}
```

(*End definition for* `\ekvd@type@code`, `\ekvd@t@code`, *and* `\ekvd@t@ecode`.)

`\ekvd@type@default` `\ekvd@t@default` `\ekvd@t@qdefault`

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

```
\protected\long\def\ekvd@type@default#1#2#3#4%
  {%
    \ekvd@assert@arg{#1default #3}{#4}%
      {%
        \ekvifdefined\ekvd@set{#3}%
          {%
            \ekvd@assert@not@long{#1default #3}%
            \ekvd@prot\edef\ekvd@tmp
              {%
                \unexpanded\expandafter#2%
                  {\csname\ekv@name\ekvd@set{#3}\endcsname{#4}}%
              }%
            \ekvletNoVal\ekvd@set{#3}\ekvd@tmp
          }%
          {\ekvd@err@undefined@key{#3}}%
      }%
  }
\protected\def\ekvd@t@default{\ekvd@type@default{}{}}
\protected\def\ekvd@t@qdefault{\ekvd@type@default q{\expandafter\expandafter}}
```

(*End definition for* `\ekvd@type@default`, `\ekvd@t@default`, *and* `\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#1#2%
  {%
    \ekvd@assert@arg{edefault #1}{#2}%
      {%
```

```
        \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}}%
      }%
  }
```

(*End definition for* `\ekvd@t@initial`, `\ekvd@t@oinitial`, *and* `\ekvd@t@einitial`.)

`\ekvd@type@bool` `\ekvd@t@bool` `\ekvd@t@gbool` `\ekvd@t@boolTF` `\ekvd@t@gboolTF`

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\ekvdefNoVal\ekvd@set{#6}{#3\let#7#4}%
        \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{\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.*)

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

```
\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` `\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. `\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@assert@filledarg{#1box #3}{#4}%
      {%
        \ekvd@newreg#4{box}%
        \protected\ekvd@long\ekvdef\ekvd@set{#3}%
          {#2\setbox#4\hbox{\begingroup##1\endgroup}}%
      }%
  }
\protected\def\ekvd@t@box{\ekvd@type@box{}{}}
\protected\def\ekvd@t@gbox{\ekvd@type@box g\global}
```

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

`\ekvd@type@toks` `\ekvd@t@toks` `\ekvd@t@gtoks`

Similar to box, but set the toks.

```
\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{##1}}%
      }%
  }
\protected\def\ekvd@t@toks{\ekvd@type@toks{}{}}
\protected\def\ekvd@t@gtoks{\ekvd@type@toks{g}\global}
```

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

`\ekvd@type@apptoks` `\ekvd@t@apptoks` `\ekvd@t@gapptoks`

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

```
\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}}%
      }%
  }
\protected\def\ekvd@t@apptoks{\ekvd@type@apptoks{}{}}
\protected\def\ekvd@t@gapptoks{\ekvd@type@apptoks{g}\global}
```

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

`\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`

The `\ekvd@type@reg` can handle all the types for which the assignment will just be ⟨*register*⟩=⟨*value*⟩.

```
\protected\def\ekvd@type@reg#1#2#3#4#5#6#7%
  {%
    \ekvd@assert@filledarg{#1 #6}{#7}%
      {%
        \ekvd@newreg#7{#2}%
        \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\relax}
\protected\def\ekvd@t@gint{\ekvd@type@reg{gint}{count}\global{}{}}
\protected\def\ekvd@t@xint{\ekvd@type@reg{xint}{count}\global\numexpr\relax}
```

```
\protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{dimen}{}{}{}}
\protected\def\ekvd@t@edimen{\ekvd@type@reg{edimen}{dimen}{}\dimexpr\relax}
\protected\def\ekvd@t@gdimen{\ekvd@type@reg{gdimen}{dimen}\global{}{}}
\protected\def\ekvd@t@xdimen{\ekvd@type@reg{xdimen}{dimen}\global\dimexpr\relax}
\protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{skip}{}{}{}}
\protected\def\ekvd@t@eskip{\ekvd@type@reg{eskip}{skip}{}\glueexpr\relax}
\protected\def\ekvd@t@gskip{\ekvd@type@reg{gskip}{skip}\global{}{}}
\protected\def\ekvd@t@xskip{\ekvd@type@reg{xskip}{skip}\global\glueexpr\relax}
```

(*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#1#2#3#4%
  {%
    \ekvd@assert@filledarg{#1store #3}{#4}%
      {%
        \unless\ifdefined#4\let#4\ekvd@empty\fi
        \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{\unexpanded{##1}}}%
      }%
  }
\protected\def\ekvd@t@store{\ekvd@type@store{}\edef}
\protected\def\ekvd@t@gstore{\ekvd@type@store{g}\xdef}
```

(*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#1#2#3#4%
  {%
    \ekvd@assert@filledarg{#1store #3}{#4}%
      {%
        \ekvd@newlet#4\ekvd@empty
        \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{##1}}%
      }%
  }
\protected\def\ekvd@t@estore{\ekvd@type@estore{e}\edef}
\protected\def\ekvd@t@xstore{\ekvd@type@estore{x}\xdef}
```

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

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

`meta` sets up things such that another instance of `\ekvset` will be run on the argument, with the same ⟨*set*⟩.

```
\protected\long\def\ekvd@type@meta#1#2#3#4#5%
  {%
    \ekvd@assert@filledarg{#1meta #4}{#5}%
      {%
        \edef\ekvd@tmp{\ekvd@set}%
        \expandafter\ekvd@type@meta@a\expandafter{\ekvd@tmp}{#5}{#3}%
        #2\ekvd@set{#4}\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\expandafter
  }
\protected\long\def\ekvd@type@meta@c#1#2%
  {%
    \ekvd@prot\ekvd@long\def\ekvd@tmp#2{#1}%
  }
\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.*)

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

`smeta` is pretty similar to `meta`, but needs two arguments inside of ⟨*value*⟩, such that the first is the ⟨*set*⟩ for which the sub-`\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}%
      {%
        \ekvd@type@meta@a#5{#3}%
        #2\ekvd@set{#4}\ekvd@tmp
      }%
  }
\protected\def\ekvd@t@smeta{\ekvd@type@smeta{}\ekvlet{##1}}
\protected\long\def\ekvd@t@snmeta#1#2%
  {%
    \ekvd@assert@not@long{snmeta #1}%
    \ekvd@type@smeta n\ekvletNoVal{}{#1}{#2}%
  }
```

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

`\ekvd@type@choice` `\ekvd@populate@choice` `\ekvd@populate@choice@` `\ekvd@populate@choice@noarg` `\ekvd@choice@prefix` `\ekvd@choice@prefix@` `\ekvd@choice@p@protected` `\ekvd@choice@p@protect` `\ekvd@choice@p@long` `\ekvd@choice@p@long@` `\ekvd@t@choice`

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}%
    \ekvd@prot\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 `\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@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}%
      {%
        \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{#2}%
          {\ekvd@choice@prefix#2\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}%
    \ekvd@ifspace{#1}%
  }
\protected\def\ekvd@choice@p@long@#1%
```

```
  {%
    \expandafter\ekvd@err@no@long\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}
```

(*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`.)

`\ekvd@newlet` `\ekvd@newreg` 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`.)

`\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.

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

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

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

```
\long\def\ekvd@assert@arg#1#2%
  {%
    \ekvd@ifnoarg{#2}%
      {\ekvd@err@missing@definition{#1}}%
  }
\long\def\ekvd@ifnoarg#1%
  {%
    \ifx\ekvd@noarg@mark#1%
      \ekvd@fi@firstoftwo
    \fi
    \@secondoftwo
  }
```

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

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

```
\long\def\ekvd@assert@filledarg#1#2%
  {%
    \ekvd@ifnoarg@or@empty{#2}%
      {\ekvd@err@missing@definition{#1}}%
```

```
  }
\long\def\ekvd@ifnoarg@or@empty#1%
  {%
    \ekvd@ifnoarg{#1}%
      \@firstoftwo
      {\ekv@ifempty{#1}}%
  }
```

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

`\ekvd@assert@not@long` `\ekvd@assert@not@protected`

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.

```
\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
  }
```

(*End definition for* `\ekvd@assert@not@long` *and* `\ekvd@assert@not@protected`.)

`\ekvd@ifspace` `\ekvd@ifspace@`

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@`.)

2.2.4 Messages

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

`\ekvd@err@missing@definition` `\ekvd@err@missing@prefix` `\ekvd@err@undefined@prefix` `\ekvd@err@undefined@key` `\ekvd@err@no@protected` `\ekvd@err@no@long`

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

```
\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 `\unexpanded{#1}'}}
\protected\def\ekvd@err@undefined@key#1%
  {\errmessage{expkv-def Error: Undefined key `\unexpanded{#1}'}}
\protected\def\ekvd@err@no@protected#1%
  {%
```

```
    \errmessage
      {expkv-def Error: prefix ‘protected’ not accepted for ‘\unexpanded{#1}’}%
  }
\protected\def\ekvd@err@no@long#1%
  {%
    \errmessage
      {expkv-def Error: prefix ‘long’ not accepted for ‘\unexpanded{#1}’}%
  }
```

(*End definition for* `\ekvd@err@missing@definition` *and others.*)

`\ekvd@err@choice@invalid`
`\ekvd@err@choice@invalid@`
`\ekvd@choice@name`
`\ekvd@err`

The expandable error messages use `\ekvd@err`, which is just like `\ekv@err` from **expkv** 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%
  {%
    \ekvd@err@choice@invalid@#1\ekv@stop
  }
\begingroup
\catcode40=8
\catcode41=8
\@firstofone{\endgroup
\def\ekvd@choice@name#1#2#3%
  {%
    ekvd#1(#2)#3%
  }
\def\ekvd@err@choice@invalid@ ekvd#1(#2)#3\ekv@stop%
  {%
    \ekvd@err{invalid choice ‘#3’ (‘#2’, set ‘#1’)}%
  }
}
\begingroup
\edef\ekvd@err
  {%
    \endgroup
    \unexpanded{\long\def\ekvd@err}##1%
      {%
        \unexpanded{\expandafter\ekv@err@\@firstofone}%
        {\unexpanded\expandafter{\csname ! expkv-def Error:\endcsname}##1.}%
        \unexpanded{\ekv@stop}%
      }%
  }
\ekvd@err
```

(*End definition for* `\ekvd@err@choice@invalid` *and others.*)

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

```
\catcode‘\@=\ekvd@tmp
```

Index

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

X