expkv|def

a key-defining frontend for expkv

Jonathan P. Spratte*

2021-09-20 v0.8c

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, as a LATEX package, and as a ConTEXt module. It'll automatically load expkv in the same mode as well. To use it, just use one of

```
\input expkv-def          % plainTeX
\usepackage{expkv-def}    % LaTeX
\usemodule[expkv-def]     % ConTeXt
```

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 two p-type prefixes `long` and `protected` are pretty simple by nature, so their description is pretty simple. They affect the ⟨*key*⟩ at use-time, so omitting `long` doesn't mean that a ⟨*definition*⟩ can't contain a `\par` token, only that the ⟨*key*⟩ will not accept

a \par in ⟨*value*⟩. On the other hand `new` and `also` might be simple on first sight as well, but their rules are a bit more complicated.

`also`

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

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

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

```
\ekvdefinekeys{also-example}
  {
    bool key       = \ifmybool
   ,also code key = \domystuff{#1}
  }
```

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

`protected` `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.

`new`

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

```
\ekvdefinekeys{new-example}
  {
    code key       = \domystuffwitharg{#1}
   ,new  noval key = \domystuffwithoutarg
  }
```

1.2.2 t-Prefixes

Since the p-type prefixes apply to some of the t-prefixes automatically but sometimes one might be disallowed we need some way to highlight this behaviour. In the following

an enforced prefix will be printed black (protected), allowed prefixes will be grey (protected), and disallowed prefixes will be red (protected). This will be put flush-right in the syntax showing line.

---

code
ecode

`code` ⟨*key*⟩ `=` {⟨*definition*⟩} — new also 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*⟩} — new also 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
odefault
fdefault
edefault

`default` ⟨*key*⟩ `=` {⟨*definition*⟩} — new also 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*⟩. `odefault` is just another name for `qdefault`. The `fdefault` version will expand the key code until a non-expandable token or a space is found, a space would be gobbled.[1] The `edefault` on the other hand fully expands the ⟨*key*⟩-code with ⟨*definition*⟩ as its argument inside of an `\edef`.

---

initial
oinitial
finitial
einitial

`initial` ⟨*key*⟩ `=` {⟨*value*⟩} — new also protected long
`initial` ⟨*key*⟩

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. `finitial` will expand ⟨*value*⟩ until a non-expandable token or a space is found, a space would be gobbled.[2]

If you don't provide a value (and no equals sign) a `noval` ⟨*key*⟩ of the same name is called once (or, if you specified a `default` for a value taking key that would be used).

[1] For those familiar with TEX-coding: This uses a `\romannumeral`-expansion.
[2] Again using `\romannumeral`.

bool
gbool
boolTF
gboolTF

`bool` ⟨*key*⟩ `=` ⟨*cs*⟩ — new also 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). Internally a `bool` ⟨*key*⟩ is the same as a `choice` key which is set up to handle `true` and `false` as choices.

invbool
ginvbool
invboolTF
ginvboolTF

`bool` ⟨*key*⟩ `=` ⟨*cs*⟩ — new also protected long

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

boolpair
gboolpair
boolpairTF
gboolpairTF

`boolpair` ⟨*key*⟩ `=` ⟨cs_1⟩⟨cs_2⟩ — new also protected long

The `boolpair` key type behaves like both `bool` and `invbool`, the ⟨cs_1⟩ will be set to the meaning according to the rules of `bool`, and ⟨cs_2⟩ will be set to the opposite.

store
estore
gstore
xstore

`store` ⟨*key*⟩ `=` ⟨*cs*⟩ — new also **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*⟩ — new also **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*⟩ — new also **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*⟩ — new also 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*⟩ — new also 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*⟩ — new also 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*⟩ — new also 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*⟩ — new also 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*⟩, ...} — new also 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*⟩, ...} — new also 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*⟩, ...} — new also 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*⟩, ...} — new also 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⟩}` new also 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⟩, ...}` new also 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 (but see the `unknown-choice` prefix). You can add additional choices after the ⟨*key*⟩ was created by using `choice` again for the same ⟨*key*⟩, redefining choices is possible the same way, but there is no interface to remove certain choices.

unknown-choice `unknown-choice ⟨key⟩ = {⟨definition⟩}` new also protected long

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

unknown␣code `unknown code = {⟨definition⟩}` new also protected long

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

With `unknown code` the ⟨*definition*⟩ is used for unknown keys which were provided a value (so corresponds to `\ekvdefunknown`), you can access the key name with `#1` and the value with `#2`.[3]

unknown␣noval `unknown noval = {⟨definition⟩}` new also protected long

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

unknown␣redirect-code `unknown redirect-code = {⟨set-list⟩}` new also protected long

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

[3] There is some trickery involved to get this more intuitive argument order without any performance hit if you compare this to `\ekvdefunknown` directly.

unknown␣redirect-noval

`unknown redirect-noval = {⟨set-list⟩}` new also protected long

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

unknown␣redirect

`unknown redirect = {⟨set-list⟩}` new also protected long

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

1.3 Bugs

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

1.4 Example

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

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

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

```
\ekvdefinekeys{ex}
  {
     data  name  = \Pname
    ,data  age   = \Page
    ,dataT hobby = \Phobby
  }
\newcommand\Person[1]
  {%
    \begingroup
    \ekvset{ex}{#1}%
    \begin{description}
      \item[\Pname{}]{\errmessage{A person requires a name}}]
      \item[Age] \Page{\textit}{\errmessage{A person requires an age}}
      \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 ConTeXt module

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

2.3 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{0.8c}
\def\ekvdDate{2021-09-20}
```

(*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
\ekvd@ifalso

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

```
\def\ekvd@empty{}
\protected\def\ekvd@clear@prefixes
  {%
    \let\ekvd@long\ekvd@empty
    \let\ekvd@prot\ekvd@empty
    \let\ekvd@ifalso\@secondoftwo
    \long\def\ekvd@ifnew##1##2##3{##3}%
  }
\ekvd@clear@prefixes
```

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

\ekvdefinekeys

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@arg
  }
```

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

\ekvd@noarg
\ekvd@arg
\ekvd@handle

\ekvd@noarg and \ekvd@arg store whether there was a value in the ⟨*key*⟩=⟨*value*⟩ pair. \ekvd@handle has to test whether there is a space inside the key and if so calls the prefix grabbing routine, else we throw an error and ignore the key.

```
\protected\long\def\ekvd@noarg#1%
  {%
    \let\ekvd@ifnoarg\@firstoftwo
    \expandafter\ekvd@handle\detokenize{#1}\ekvd@stop{}%
  }
\protected\long\def\ekvd@arg#1%
  {%
    \let\ekvd@ifnoarg\@secondoftwo
    \expandafter\ekvd@handle\detokenize{#1}\ekvd@stop
  }
\protected\long\def\ekvd@handle#1\ekvd@stop#2%
  {%
    \ekvd@clear@prefixes
    \edef\ekvd@cur{\detokenize{#1}}%
    \ekvd@ifspace{#1}%
      {\ekvd@prefix\ekv@mark#1\ekv@stop{#2}}%
      \ekvd@err@missing@type
  }
```

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

\ekvd@prefix
\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\ekvd@prefix@after@p{#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}%
      {\ekvd@err@missing@type\@gobble}%
  }
```

(*End definition for* \ekvd@prefix@after@p.)

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

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

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

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

2.3.1 Key Types

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

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

```
\protected\def\ekvd@type@set#1#2%
  {%
    \ekvd@assert@not@long
    \ekvd@assert@not@protected
    \ekvd@ifnew{NoVal}{#1}%
      {%
        \ekv@ifempty{#2}%
          {\ekvd@err@missing@definition}%
          {%
            \ekvd@ifalso
              {%
```

```
                \ekv@expargtwice{\ekvd@add@noval{#1}}%
                  {\ekvchangeset{#2}}%
                  \ekvd@assert@not@protected@also
              }%
              {\ekv@expargtwice{\ekvdefNoVal\ekvd@set{#1}}{\ekvchangeset{#2}}}%
          }%
      }%
  }
\protected\def\ekvd@t@set#1#2%
  {%
    \ekvd@ifnoarg
      {\ekvd@type@set{#1}{#1}}%
      {\ekvd@type@set{#1}{#2}}%
  }
```

(*End definition for* `\ekvd@type@set` *and* `\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%
  {%
    \ekvd@ifnew{NoVal}{#2}%
      {%
        \ekvd@assert@arg
          {%
            \ekvd@assert@not@long
            \ekvd@prot#1\ekvd@tmp{#3}%
            \ekvd@ifalso
              {\ekv@exparg{\ekvd@add@noval{#2}}\ekvd@tmp{}}%
              {\ekvletNoVal\ekvd@set{#2}\ekvd@tmp}%
          }%
      }%
  }
\protected\def\ekvd@t@noval{\ekvd@type@noval\def}
\protected\def\ekvd@t@enoval{\ekvd@type@noval\edef}
```

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

`\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%
  {%
    \ekvd@ifnew{}{#2}%
      {%
        \ekvd@assert@arg
          {%
            \ekvd@prot\ekvd@long#1\ekvd@tmp##1{#3}%
            \ekvd@ifalso
              {\ekv@exparg{\ekvd@add@val{#2}}{\ekvd@tmp{##1}}{}}%
              {\ekvlet\ekvd@set{#2}\ekvd@tmp}%
          }%
```

```
      }%
  }
\protected\def\ekvd@t@code{\ekvd@type@code\def}
\protected\def\ekvd@t@ecode{\ekvd@type@code\edef}
```

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

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

`\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
      {%
        \ekvifdefined\ekvd@set{#3}%
          {%
            \ekvd@assert@not@new
            \ekvd@assert@not@long
            \ekvd@prot\edef\ekvd@tmp
              {%
                \ekv@unexpanded\expandafter#1%
                  {#2\csname\ekv@name\ekvd@set{#3}\endcsname{#4}}%
              }%
            \ekvd@ifalso
              {\ekv@exparg{\ekvd@add@noval{#3}}\ekvd@tmp{}}%
              {\ekvletNoVal\ekvd@set{#3}\ekvd@tmp}%
          }%
          {\ekvd@err@undefined@key{#3}}%
      }%
  }
\protected\def\ekvd@t@default{\ekvd@type@default{}{}}
\protected\def\ekvd@t@qdefault{\ekvd@type@default{\expandafter\expandafter}{}}
\let\ekvd@t@odefault\ekvd@t@qdefault
\protected\def\ekvd@t@fdefault{\ekvd@type@default{}{\romannumeral`\^^@}}
```

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

`\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
      {%
        \ekvifdefined\ekvd@set{#1}%
          {%
            \ekvd@assert@not@new
            \ekvd@assert@not@long
            \ekvd@prot\edef\ekvd@tmp
              {\csname\ekv@name\ekvd@set{#1}\endcsname{#2}}%
            \ekvd@ifalso
              {\ekv@exparg{\ekvd@add@noval{#1}}\ekvd@tmp{}}%
              {\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@finitial`
`\ekvd@t@einitial`

```
\long\def\ekvd@type@initial#1#2#3#4%
  {%
    \ekvd@assert@not@new
    \ekvd@assert@not@also
    \ekvd@assert@not@long
    \ekvd@assert@not@protected
    \ekvd@ifnoarg
      {%
        \ekvifdefinedNoVal\ekvd@set{#3}%
          {\csname\ekv@name\ekvd@set{#3}N\endcsname}%
          {\ekvd@err@undefined@noval{#3}}%
      }%
      {%
        \ekvifdefined\ekvd@set{#3}%
          {%
            #1{#2#4}%
            \csname\ekv@name\ekvd@set{#3}\expandafter\endcsname\expandafter
              {\ekvd@tmp}%
          }%
          {\ekvd@err@undefined@key{#3}}%
      }%
  }
\def\ekvd@t@initial{\ekvd@type@initial{\def\ekvd@tmp}{}}
\def\ekvd@t@oinitial{\ekvd@type@initial{\ekv@exparg{\def\ekvd@tmp}}{}}
\def\ekvd@t@einitial{\ekvd@type@initial{\edef\ekvd@tmp}{}}
\def\ekvd@t@finitial
  {\ekvd@type@initial{\ekv@exparg{\def\ekvd@tmp}}{\romannumeral`\^^@}}
```

(*End definition for* `\ekvd@t@initial` *and others.*)

`\ekvd@type@bool`
`\ekvd@t@bool`
`\ekvd@t@gbool`
`\ekvd@t@boolTF`
`\ekvd@t@gboolTF`
`\ekvd@t@invbool`
`\ekvd@t@ginvbool`
`\ekvd@t@invboolTF`
`\ekvd@t@ginvboolTF`

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%
  {%
    \ekvd@ifnew{}{#4}%
      {%
        \ekvd@ifnew{NoVal}{#4}%
          {%
            \ekvd@assert@filledarg{#5}%
              {%
                \ekvd@newlet#5#3%
                \ekvd@type@choice{#4}%
                \protected\ekvdefNoVal\ekvd@set{#4}{#1\let#5#2}%
                \protected\expandafter\def
                  \csname\ekvd@choice@name\ekvd@set{#4}{true}\endcsname
```

```
                  {#1\let#5#2}%
                \protected\expandafter\def
                  \csname\ekvd@choice@name\ekvd@set{#4}{false}\endcsname
                  {#1\let#5#3}%
              }%
          }%
      }%
  }
\protected\def\ekvd@t@bool{\ekvd@type@bool{}\iftrue\iffalse}
\protected\def\ekvd@t@gbool{\ekvd@type@bool\global\iftrue\iffalse}
\protected\def\ekvd@t@boolTF{\ekvd@type@bool{}\@firstoftwo\@secondoftwo}
\protected\def\ekvd@t@gboolTF{\ekvd@type@bool\global\@firstoftwo\@secondoftwo}
\protected\def\ekvd@t@invbool{\ekvd@type@bool{}\iffalse\iftrue}
\protected\def\ekvd@t@ginvbool{\ekvd@type@bool\global\iffalse\iftrue}
\protected\def\ekvd@t@invboolTF{\ekvd@type@bool{}\@secondoftwo\@firstoftwo}
\protected\def\ekvd@t@ginvboolTF
  {\ekvd@type@bool\global\@secondoftwo\@firstoftwo}
```

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

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

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

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

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

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

```
\protected\def\ekvd@type@data#1#2#3#4#5#6%
  {%
    \ekvd@ifnew{}{#5}%
      {%
        \ekvd@assert@filledarg{#6}%
          {%
            \ekvd@newlet#6#1%
            \ekvd@ifalso
              {%
                \let\ekvd@prot\protected
                \ekvd@add@val{#5}{\long#2#6####1#3{####1{#4}}}{}%
              }%
              {%
                \protected\ekvd@long\ekvdef\ekvd@set{#5}%
                  {\long#2#6####1#3{####1{#4}}}%
              }%
          }%
      }%
  }
\protected\def\ekvd@t@data
  {\ekvd@type@data\@secondoftwo\edef{####2}{\ekv@unexpanded{##1}}}
\protected\def\ekvd@t@edata{\ekvd@type@data\@secondoftwo\edef{####2}{##1}}
\protected\def\ekvd@t@gdata
  {\ekvd@type@data\@secondoftwo\xdef{####2}{\ekv@unexpanded{##1}}}
\protected\def\ekvd@t@xdata{\ekvd@type@data\@secondoftwo\xdef{####2}{##1}}
\protected\def\ekvd@t@dataT
  {\ekvd@type@data\@gobble\edef{}{\ekv@unexpanded{##1}}}
\protected\def\ekvd@t@edataT{\ekvd@type@data\@gobble\edef{}{##1}}
\protected\def\ekvd@t@gdataT
  {\ekvd@type@data\@gobble\xdef{}{\ekv@unexpanded{##1}}}
\protected\def\ekvd@t@xdataT{\ekvd@type@data\@gobble\xdef{}{##1}}
```

(*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%
  {%
    \ekvd@ifnew{}{#2}%
      {%
        \ekvd@assert@filledarg{#3}%
          {%
            \ekvd@newreg#3{box}%
            \ekvd@ifalso
              {%
                \let\ekvd@prot\protected
                \ekvd@add@val{#2}{#1\setbox#3\hbox{\begingroup##1\endgroup}}{}%
              }%
              {%
```

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

(*End definition for* `\ekvd@type@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%
  {%
    \ekvd@ifnew{}{#2}%
      {%
        \ekvd@assert@filledarg{#3}%
          {%
            \ekvd@newreg#3{toks}%
            \ekvd@ifalso
              {%
                \let\ekvd@prot\protected
                \ekvd@add@val{#2}{#1#3{##1}}{}%
              }%
              {\protected\ekvd@long\ekvdef\ekvd@set{#2}{#1#3{##1}}}%
          }%
      }%
  }
\protected\def\ekvd@t@toks{\ekvd@type@toks{}}
\protected\def\ekvd@t@gtoks{\ekvd@type@toks\global}
```

(*End 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%
  {%
    \ekvd@ifnew{}{#2}%
      {%
        \ekvd@assert@filledarg{#3}%
          {%
            \ekvd@newreg#3{toks}%
            \ekvd@ifalso
              {%
                \let\ekvd@prot\protected
                \ekvd@add@val{#2}{#1#3\expandafter{\the#3##1}}{}%
              }%
              {%
                \protected\ekvd@long\ekvdef\ekvd@set{#2}%
                  {#1#3\expandafter{\the#3##1}}%
              }%
          }%
      }%
  }
```

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

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

`\ekvd@type@reg` `\ekvd@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%
  {%
    \ekvd@ifnew{}{#5}%
      {%
        \ekvd@assert@filledarg{#6}%
          {%
            \ekvd@newreg#6{#1}%
            \ekvd@ifalso
              {%
                \let\evkd@prot\protected
                \ekvd@add@val{#5}{#2#6=#3##1#4\relax}{}%
              }%
              {\protected\ekvd@long\ekvdef\ekvd@set{#5}{#2#6=#3##1#4\relax}}%
          }%
      }%
  }
\protected\def\ekvd@t@int{\ekvd@type@reg{count}{}{}{}}
\protected\def\ekvd@t@eint{\ekvd@type@reg{count}{}\numexpr\relax}
\protected\def\ekvd@t@gint{\ekvd@type@reg{count}\global{}{}}
\protected\def\ekvd@t@xint{\ekvd@type@reg{count}\global\numexpr\relax}
\protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{}{}{}}
\protected\def\ekvd@t@edimen{\ekvd@type@reg{dimen}{}\dimexpr\relax}
\protected\def\ekvd@t@gdimen{\ekvd@type@reg{dimen}\global{}{}}
\protected\def\ekvd@t@xdimen{\ekvd@type@reg{dimen}\global\dimexpr\relax}
\protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{}{}{}}
\protected\def\ekvd@t@eskip{\ekvd@type@reg{skip}{}\glueexpr\relax}
\protected\def\ekvd@t@gskip{\ekvd@type@reg{skip}\global{}{}}
\protected\def\ekvd@t@xskip{\ekvd@type@reg{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@ifnew{}{#3}%
      {%
        \ekvd@assert@filledarg{#4}%
          {%
            \ekvd@newlet#4\ekvd@empty
            \ekvd@ifalso
              {%
                \let\ekvd@prot\protected
                \ekvd@add@val{#3}{#1#4{#2}}{}%
              }%
              {\protected\ekvd@long\ekvdef\ekvd@set{#3}{#1#4{#2}}}%
          }%
```

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

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

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

(*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#6#7%
  {%
    \ekvd@ifnew{#1}{#6}%
      {%
        \ekvd@assert@twoargs{#7}%
          {%
```

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

(*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%
  {%
    \ekvd@assert@not@long
    \ekvd@prot\edef\ekvd@tmp##1%
      {\ekv@unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#1}{##1}}}%
    \ekvd@ifalso
      {%
        \ekvd@assert@val{#1}%
          {%
            \ekvd@if@not@already@choice{#1}%
              {%
                \ekv@exparg
                  {%
                    \expandafter\ekvd@add@aux
                      \csname\ekv@name\ekvd@set{#1}\endcsname{{##1}}{#1}%
                  }%
                  {\ekvd@tmp{##1}}%
                  {\ekvd@long\ekvdef}\ekvd@assert@not@long@also
              }%
          }%
      }%
      {\ekvlet\ekvd@set{#1}\ekvd@tmp}%
  }
```

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

```
\protected\def\ekvd@populate@choice
  {%
    \ekvparse\ekvd@populate@choice@noarg\ekvd@populate@choice@
  }
\protected\long\def\ekvd@populate@choice@noarg#1%
  {%
    \expandafter\ekvd@err@missing@definition@msg\expandafter{\ekvd@cur : #1}%
  }
```

\ekvd@populate@choice@ runs the prefix-test, if there is none we can directly define the choice, for that \ekvd@set@choice will expand to the current choice-key's name, which will have been defined by \ekvd@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@msg\expandafter{\ekvd@cur : #1}%
      {%
        \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@invalid@p#1\ekvd@ifspace#2%
  {%
    \expandafter\ekvd@choice@invalid@p@\expandafter{\ekv@gobble@mark#2}{#1}%
    \ekvd@ifspace{#2}%
  }
\protected\def\ekvd@choice@invalid@p@#1#2%
  {%
    \expandafter\ekvd@err@no@prefix@msg\expandafter{\ekvd@cur : #2 #1}{#2}%
  }
\protected\def\ekvd@choice@p@long{\ekvd@choice@invalid@p{long}}%
\protected\def\ekvd@choice@p@also{\ekvd@choice@invalid@p{also}}%
\protected\def\ekvd@choice@p@new{\ekvd@choice@invalid@p{new}}%
```

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@ifnew{}{#1}%
      {%
        \ekvd@assert@arg
          {%
            \ekvd@type@choice{#1}%
            \def\ekvd@set@choice{#1}%
            \ekvd@populate@choice{#2}%
          }%
      }%
  }
```

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

`\ekvd@t@unknown-choice`

```
\protected\long\expandafter\def\csname ekvd@t@unknown-choice\endcsname#1#2%
  {%
    \ekvd@assert@new@for@name{\ekvd@unknown@choice@name\ekvd@set{#1}}%
      {%
        \ekvd@assert@arg
          {%
            \ekvd@assert@not@long
            \ekvd@assert@not@also
            \ekvd@prot\expandafter
            \def\csname\ekvd@unknown@choice@name\ekvd@set{#1}\endcsname##1{#2}%
          }%
      }%
  }
```

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

`\ekvd@t@unknown` `\ekvd@type@unknown@code` `\ekvd@type@unknown@noval`

The `unknown` type has different subtypes which would be the key names for other types. It is first checked whether that subtype is defined, if it isn't throw an error, else use that subtype.

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

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

```
\protected\long\def\ekvd@type@unknown@noval#1%
  {%
    \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}uN}%
      {%
        \ekvd@assert@arg
          {%
            \ekvd@assert@not@also
            \ekvd@assert@not@long
            \ekvd@prot\ekvdefunknownNoVal\ekvd@set{#1}%
```

```
          }%
      }%
  }
```

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

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

(*End definition for* \ekvd@t@unknown, \ekvd@type@unknown@code, *and* \ekvd@type@unknown@noval.)

\ekvd@type@unknown@redirect
\ekvd@type@unknown@redirect-code
\ekvd@type@unknown@redirect-noval

The unknown redirect types also just forward to \ekvredirectunknown after asserting some prefixes.

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

```
            \ekvd@assert@not@protected
            \ekvd@assert@not@long
            \expandafter\ekvredirectunknownNoVal\expandafter{\ekvd@set}{#1}%
          }%
      }%
  }
```

(*End definition for* `\ekvd@type@unknown@redirect`, `\ekvd@type@unknown@redirect-code`, *and* `\ekvd@type@unknown@redirect-n`)

2.3.2 Key Type Helpers

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

`\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@`.)

2.3.3 Handling `also`

`\ekvd@add@val` `\ekvd@add@noval` `\ekvd@add@aux` `\ekvd@add@aux@`

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

```
    \ekvd@extract@prefixes#1%
    \expandafter\ekvd@add@aux@\expandafter{#1#2}%
  }
\protected\long\def\ekvd@add@aux@#1#2#3#4#5%
  {%
    #5%
    \ekvd@prot#4\ekvd@set{#2}{#1#3}%
  }
```

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

`\ekvd@extract@prefixes` `\ekvd@extract@prefixes@` `\ekvd@extract@prefixes@long` `\ekvd@extract@prefixes@prot`

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

```
\protected\def\ekvd@extract@prefixes#1%
  {%
    \expandafter\ekvd@extract@prefixes@\meaning#1\ekvd@stop
  }
```

In the following definition `#1` will get replaced by `macro:`, `#2` by `\long` and `#3` by `\protected` (in each, all tokens will have category other). This allows us to parse the `\meaning` of a macro for those strings.

```
\protected\def\ekvd@extract@prefixes@#1#2#3%
  {%
    \protected\def\ekvd@extract@prefixes@##1#1##2\ekvd@stop
      {%
        \ekvd@extract@prefixes@long
          ##1\ekvd@mark\@firstofone#2\ekvd@mark\@gobble\ekvd@stop
          {\let\ekvd@long\long}%
        \ekvd@extract@prefixes@prot
          ##1\ekvd@mark\@firstofone#3\ekvd@mark\@gobble\ekvd@stop
          {\let\ekvd@prot\protected}%
      }%
    \protected\def\ekvd@extract@prefixes@long##1#2##2\ekvd@mark##3##4\ekvd@stop
      {##3}%
    \protected\def\ekvd@extract@prefixes@prot##1#3##2\ekvd@mark##3##4\ekvd@stop
      {##3}%
  }
```

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

```
\begingroup
\edef\ekvd@tmp
  {%
    \endgroup
    \ekvd@extract@prefixes@
      {\detokenize{macro:}}%
      {\string\long}%
      {\string\protected}%
  }
\ekvd@tmp
```

(*End definition for* `\ekvd@extract@prefixes` *and others.*)

2.3.4 Tests

\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%
  {%
    \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\let#1#2}%
  }
\protected\def\ekvd@newreg#1#2%
  {%
    \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\csname new#2\endcsname#1}%
  }
```

(*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%
  {%
    \ekvd@ifnottwoargs{#1}{\ekvd@err@missing@definition}%
  }
\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@val
\ekvd@assert@val@
\ekvd@assert@noval
\ekvd@assert@noval@
\ekvd@extract@args
\ekvd@extracted@args
\ekvd@one@arg@string

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

```
\protected\def\ekvd@assert@val#1%
  {%
    \ekvifdefined\ekvd@set{#1}%
      {\expandafter\ekvd@assert@val@\csname\ekv@name\ekvd@set{#1}\endcsname}%
      {%
        \ekvifdefinedNoVal\ekvd@set{#1}%
          \ekvd@err@add@val@on@noval
          {\ekvd@err@undefined@key{#1}}%
        \@gobble
      }%
  }
\protected\def\ekvd@assert@val@#1%
  {%
    \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
    \unless\ifx\ekvd@extracted@args\ekvd@one@arg@string
      \ekvd@err@unsupported@arg
    \fi
    \@firstofone
  }%
\protected\def\ekvd@assert@noval#1%
```

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

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

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

There is no need to actually define `\ekvd@ifnoarg` here, as it will be set by either `\ekvd@arg` or `\ekvd@noarg`.

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

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

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

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

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

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

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

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

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

`\ekvd@if@not@already@choice`
`\ekvd@if@not@already@choice@a`
`\ekvd@if@not@already@choice@b`

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

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

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

`\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@`.)

2.3.5 Messages

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

`\ekvd@errm`
`\ekvd@err@missing@definition`
`\ekvd@err@missing@definition@msg`
`\ekvd@err@missing@type`
`\ekvd@err@undefined@prefix`
`\ekvd@err@undefined@key`
`\ekvd@err@no@prefix`
`\ekvd@err@no@prefix@msg`
`\ekvd@err@no@prefix@also`
`\ekvd@err@add@val@on@noval`
`\ekvd@err@add@noval@on@val`
`\ekvd@err@unsupported@arg`
`\ekvd@err@not@new`

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

```
\protected\def\ekvd@errm#1{\errmessage{expkv-def Error: #1}}
\protected\def\ekvd@err@missing@definition
  {\ekvd@errm{Missing definition for key ‘\ekvd@cur’}}
\protected\def\ekvd@err@missing@definition@msg#1%
  {\ekvd@errm{Missing definition for key ‘\ekv@unexpanded{#1}’}}
\protected\def\ekvd@err@missing@type
  {\ekvd@errm{Missing type prefix for key ‘\ekvd@cur’}}
\protected\def\ekvd@err@undefined@prefix#1%
  {%
    \ekvd@errm
      {%
        Undefined prefix ‘\ekv@unexpanded{#1}’ found while processing
        ‘\ekvd@cur’%
      }%
  }
\protected\def\ekvd@err@undefined@key#1%
  {%
    \ekvd@errm
      {Undefined key ‘\ekv@unexpanded{#1}’ found while processing ‘\ekvd@cur’}%
  }
\protected\def\ekvd@err@undefined@noval#1%
  {%
    \ekvd@errm
      {%
        Undefined noval key ‘\unexpanded{#1}’ found while processing
        ‘\ekvd@cur’%
      }%
  }
\protected\def\ekvd@err@no@prefix#1%
  {\ekvd@errm{prefix ‘#1’ not accepted in ‘\ekvd@cur’}}
\protected\def\ekvd@err@no@prefix@msg#1#2%
  {\ekvd@errm{prefix ‘#2’ not accepted in ‘\ekv@unexpanded{#1}’}}
\protected\def\ekvd@err@no@prefix@also#1%
  {\ekvd@errm{‘\ekvd@cur’ not allowed with a ‘#1’ key}}
\protected\def\ekvd@err@add@val@on@noval
  {\ekvd@errm{‘\ekvd@cur’ not allowed with a NoVal key}}
\protected\def\ekvd@err@add@noval@on@val
  {\ekvd@errm{‘\ekvd@cur’ not allowed with a value taking key}}
\protected\def\ekvd@err@unsupported@arg\fi\@firstofone#1%
  {%
    \fi
    \ekvd@errm
      {%
        Existing key-macro has the unsupported argument string
        ‘\ekvd@extracted@args’ for key ‘\ekvd@cur’%
      }%
```

```
  }
\protected\def\ekvd@err@not@new
  {\ekvd@errm{The key for `\ekvd@cur' is already defined}}
\protected\long\def\ekvd@err@misused@unknown
  {\ekvd@errm{Misuse of the unknown type found while processing `\ekvd@cur'}}
```

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

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

`\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@unknown@choice@name#1#2%
  {%
    ekvd:u:#1(#2)%
  }
\def\ekvd@err@choice@invalid@ ekvd#1(#2)#3\ekv@stop%
  {%
    \ekv@ifdefined{\ekvd@unknown@choice@name{#1}{#2}}%
      {\csname\ekvd@unknown@choice@name{#1}{#2}\endcsname{#3}}%
      {\ekvd@err{invalid choice `#3' for `#2' in set `#1'}}%
  }
}
```

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

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

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

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

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.

U

W

X