

exp_kv_{DEF}

a key-defining frontend for exp_kv

Jonathan P. Spratte*

2021-05-24 v0.8a

Abstract

exp_kv_{DEF} provides a small $\langle key \rangle = \langle value \rangle$ interface to define keys for exp_kv. Key-types are declared using prefixes, similar to static typed languages. The stylised name is exp_kv_{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

1	Documentation	2
1.1	Macros	2
1.2	Prefixes	2
1.2.1	p-Prefixes	2
1.2.2	t-Prefixes	3
1.3	Bugs	7
1.4	Example	8
1.5	License	9
2	Implementation	10
2.1	The L ^A T _E X Package	10
2.2	The Generic Code	10
2.2.1	Key Types	12
2.2.2	Key Type Helpers	24
2.2.3	Handling also	25
2.2.4	Tests	26
2.2.5	Messages	29
	Index	32

*jspratte@yahoo.de

1 Documentation

Since the trend for the last couple of years goes to defining keys for a $\langle key \rangle = \langle value \rangle$ interface using a $\langle key \rangle = \langle value \rangle$ interface, I thought that maybe providing such an interface for `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 $\langle key \rangle = \langle value \rangle$ interfaces with $\langle key \rangle = \langle value \rangle$ interfaces.

Unlike many of the other established $\langle key \rangle = \langle value \rangle$ interfaces to define keys, `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 [subsection 1.2.1](#), the t-prefixes are described in [subsection 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 $\langle set \rangle$, define $\langle key \rangle$ to have definition $\langle value \rangle$. The general syntax for $\langle key \rangle$ should be

```
 $\langle prefix \rangle \langle name \rangle$ 
```

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

```
\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 $\langle key \rangle$ at use-time, so omitting `long` doesn't mean that a $\langle definition \rangle$ can't contain a `\par` token, only that the $\langle key \rangle$ 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 $\langle key \rangle = \{ \langle definition \rangle \}$	new also protected long
noval enoval	noval $\langle key \rangle = \{ \langle definition \rangle \}$	new also protected long
default qdefault odefault fdefault edefault	default $\langle key \rangle = \{ \langle definition \rangle \}$	new also protected long
initial oinitial finitial einitial	initial $\langle key \rangle = \{ \langle value \rangle \}$	new also protected long
bool gbool boolTF gboolTF	bool $\langle key \rangle = \langle cs \rangle$	new also protected long

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

The noval type defines $\langle key \rangle$ to expand to $\langle definition \rangle$. The $\langle key \rangle$ will not take a $\langle value \rangle$. enoval fully expands $\langle definition \rangle$ inside an \backslashedef .

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

With initial you can set an initial $\langle value \rangle$ for an already defined argument taking $\langle key \rangle$. It'll just call the key-macro of $\langle key \rangle$ and pass it $\langle value \rangle$. The einitial variant will expand $\langle value \rangle$ using an \backslashedef expansion prior to passing it to the key-macro and the oinitial variant will expand the first token in $\langle value \rangle$ once. finitial will expand $\langle value \rangle$ until a non-expandable token or a space is found, a space would be gobbled.²

The $\langle cs \rangle$ should be a single control sequence, such as \backslashiffoo . This will define $\langle key \rangle$ to be a boolean key, which only takes the values true or false and will throw an error for other values. If the key is used without a $\langle value \rangle$ it'll have the same effect as if you use $\langle key \rangle = \text{true}$. bool and gbool will behave like T_EX-ifs so either be \backslashiftrue or \backslashiffalse . 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 $\backslash@firstoftwo$ or $\backslash@secondoftwo$. The variants with a leading g will set the control sequence globally, the others locally. If $\langle cs \rangle$ is not yet defined it'll be initialised as the false version. Note that the initialisation is *not* done with \backslashnewif , so you will not be able to do \backslashfoottrue outside of the $\langle key \rangle = \langle value \rangle$ interface, but you could use \backslashnewif yourself. Even if the $\langle key \rangle$ will not be \backslashprotected 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 $\langle key \rangle = \text{false}$ without an issue to change the default behaviour to execute the false choice). Internally a bool $\langle key \rangle$ is the same as a choice key which is set up to handle true and false as choices.

¹For those familiar with T_EX-coding: This uses a \backslashromannumeral -expansion.

²Again using \backslashromannumeral .

invbool ginvbool invboolTF ginvboolTF	bool $\langle key \rangle = \langle cs \rangle$ new also protected long
	These are inverse boolean keys, they behave like <code>bool</code> and friends but set the opposite meaning to the macro $\langle cs \rangle$ in each case. So if <code>key=true</code> is used <code>invbool</code> will set $\langle cs \rangle$ to <code>\iffalse</code> and vice versa.
boolpair gboolpair boolpairTF gboolpairTF	boolpair $\langle key \rangle = \langle cs_1 \rangle \langle cs_2 \rangle$ new also protected long
	The <code>boolpair</code> key type behaves like both <code>bool</code> and <code>invbool</code> , the $\langle cs_1 \rangle$ will be set to the meaning according to the rules of <code>bool</code> , and $\langle cs_2 \rangle$ will be set to the opposite.
store estore gstore xstore	store $\langle key \rangle = \langle cs \rangle$ new also protected long
	The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . This will define $\langle key \rangle$ to store $\langle value \rangle$ inside of the control sequence. If $\langle cs \rangle$ isn't yet defined it will be initialised as empty. The variants behave similarly to their <code>\def</code> , <code>\edef</code> , <code>\gdef</code> , and <code>\xdef</code> counterparts, but <code>store</code> and <code>gstore</code> will allow you to store macro parameters inside of them by using <code>\unexpanded</code> .
data edata gdata xdata	data $\langle key \rangle = \langle cs \rangle$ new also protected long
	The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . This will define $\langle key \rangle$ to store $\langle value \rangle$ inside of the control sequence. But unlike the <code>store</code> type, the macro $\langle cs \rangle$ will be a switch at the same time, it'll take two arguments and if $\langle key \rangle$ was used expands to the first argument followed by $\langle value \rangle$ in braces, if $\langle key \rangle$ was not used $\langle cs \rangle$ will expand to the second argument (so behave like <code>\@secondoftwo</code>). The idea is that with this type you can define a key which should be typeset formatted. The <code>edata</code> and <code>xdata</code> variants will fully expand $\langle value \rangle$, the <code>gdata</code> and <code>xdata</code> variants will store $\langle value \rangle$ inside $\langle cs \rangle$ globally. The <code>p</code> -prefixes will only affect the key-macro, $\langle cs \rangle$ will always be expandable and <code>\long</code> .
dataT edataT gdataT xdataT	dataT $\langle key \rangle = \langle cs \rangle$ new also protected long
	Just like <code>data</code> , but instead of $\langle cs \rangle$ grabbing two arguments it'll only grab one, so by default it'll behave like <code>\@gobble</code> , and if a $\langle value \rangle$ was given to $\langle key \rangle$ the $\langle cs \rangle$ will behave like <code>\@firstofone</code> appended by $\{\langle value \rangle\}$.
int eint gint xint	int $\langle key \rangle = \langle cs \rangle$ new also protected long
	The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . An <code>int</code> key will be a TeX-count register. If $\langle cs \rangle$ isn't defined yet, <code>\newcount</code> will be used to initialise it. The <code>eint</code> and <code>xint</code> versions will use <code>\numexpr</code> to allow basic computations in their $\langle value \rangle$. The <code>gint</code> and <code>xint</code> variants set the register globally.
dimen edimen gdimen xdimen	dimen $\langle key \rangle = \langle cs \rangle$ new also protected long
	The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . This is just like <code>int</code> but uses a <code>dimen</code> register, <code>\newdimen</code> and <code>\dimexpr</code> instead.
skip eskip gskip xskip	skip $\langle key \rangle = \langle cs \rangle$ new also protected long
	The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . This is just like <code>int</code> but uses a <code>skip</code> register, <code>\newskip</code> and <code>\glueexpr</code> instead.

toks	toks $\langle key \rangle = \langle cs \rangle$	new also protected long
gtoks apptoks gapptoks	The $\langle cs \rangle$ should be a single control sequence, such as $\backslash foo$. Store $\langle value \rangle$ inside of a toks-register. The g variants use $\backslash global$, the app variants append $\langle value \rangle$ to the contents of that register. If $\langle cs \rangle$ is not yet defined it will be initialised with $\backslash newtoks$.	
box	box $\langle key \rangle = \langle cs \rangle$	new also protected long
gbox	The $\langle cs \rangle$ should be a single control sequence, such as $\backslash foo$. Typesets $\langle value \rangle$ into a $\backslash hbox$ and stores the result in a box register. The boxes are colour safe. \expkvDEF doesn't provide a vbox type.	
meta	meta $\langle key \rangle = \{ \langle key \rangle = \langle value \rangle, \dots \}$	new also protected long
	This key type can set other keys, you can access the $\langle value \rangle$ which was passed to $\langle key \rangle$ inside the $\langle key \rangle = \langle value \rangle$ list with #1. It works by calling a sub- $\backslash ekvset$ on the $\langle key \rangle = \langle value \rangle$ list, so a set key will only affect that $\langle key \rangle = \langle value \rangle$ list and not the current $\backslash ekvset$. Since it runs in a separate $\backslash ekvset$ you can't use $\backslash ekvsneak$ using keys or similar macros in the way you normally could.	
nmeta	nmeta $\langle key \rangle = \{ \langle key \rangle = \langle value \rangle, \dots \}$	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 $\langle key \rangle = \langle value \rangle$ list is static.	
smeta	smeta $\langle key \rangle = \{ \langle set \rangle \} \{ \langle key \rangle = \langle value \rangle, \dots \}$	new also protected long
	Yet another meta variant. An smeta key will take a $\langle value \rangle$ which you can access using #1, but it sets the $\langle key \rangle = \langle value \rangle$ list inside of $\langle set \rangle$, so is equal to $\backslash ekvset \{ \langle set \rangle \} \{ \langle key \rangle = \langle value \rangle, \dots \}$.	
snmeta	snmeta $\langle key \rangle = \{ \langle set \rangle \} \{ \langle key \rangle = \langle value \rangle, \dots \}$	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 $\langle key \rangle = \langle value \rangle$ list inside of $\langle set \rangle$.	
set	set $\langle key \rangle = \{ \langle set \rangle \}$	new also protected long
	This will define $\langle key \rangle$ to change the set of the current $\backslash ekvset$ invocation to $\langle set \rangle$. You can omit $\langle set \rangle$ (including the equals sign), which is the same as using $set \langle key \rangle = \{ \langle key \rangle \}$. The created set key will not take a $\langle value \rangle$. Note that just like in \expkv it'll not be checked whether $\langle set \rangle$ is defined and you'll get a low-level TeX error if you use an undefined $\langle set \rangle$.	
choice	choice $\langle key \rangle = \{ \langle value \rangle = \langle definition \rangle, \dots \}$	new also protected long
	Defines $\langle key \rangle$ to be a choice key, meaning it will only accept a limited set of values. You should define each possible $\langle value \rangle$ inside of the $\langle value \rangle = \langle definition \rangle$ list. If a defined $\langle value \rangle$ is passed to $\langle key \rangle$ the $\langle definition \rangle$ will be left in the input stream. You can make individual values protected inside the $\langle value \rangle = \langle definition \rangle$ list. By default a choice key is expandable, an undefined $\langle value \rangle$ will throw an error in an expandable way (but see the unknown-choice prefix). You can add additional choices after the $\langle key \rangle$ was created by using choice again for the same $\langle key \rangle$, redefining choices is possible the same way, but there is no interface to remove certain choices.	

`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.

`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

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

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}
}
```



```

        \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

Copyright © 2020–2021 Jonathan P. Spratte

This work may be distributed and/or modified under the conditions of the \LaTeX Project Public License (LPPL), either version 1.3c of this license or (at your option) any later version. The latest version of this license is in the file:

<http://www.latex-project.org/lppl.txt>

This work is “maintained” (as per LPPL maintenance status) by
Jonathan P. Spratte.

2 Implementation

2.1 The L^AT_EX Package

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

```
1 \RequirePackage{expkv}
2 \def\ekvd@tmp
3   {%
4     \ProvidesFile{expkv-def.tex}%
5     [\ekvdDate\space v\ekvdVersion\space a key-defining frontend for expkv]%
6   }
7 \input{expkv-def.tex}
8 \ProvidesPackage{expkv-def}%
9   [\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.

```
10 \input expkv
    We make sure that expkv-def.tex is only input once:
11 \expandafter\ifx\csname ekvdVersion\endcsname\relax
12 \else
13   \expandafter\endinput
14 \fi
```

`\ekvdVersion` We're on our first input, so let's store the version and date in a macro.

```
\ekvdDate
15 \def\ekvdVersion{0.8a}
16 \def\ekvdDate{2021-05-24}
```

(End definition for `\ekvdVersion` and `\ekvdDate`. These functions are documented on page 2.)

If the L^AT_EX 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.

```
17 \csname ekvd@tmp\endcsname
    Store the category code of @ to later be able to reset it and change it to 11 for now.
18 \expandafter\chardef\csname ekvd@tmp\endcsname=\catcode'\@
19 \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
\ekvd@long \ekvd@prot \ekvd@clear@prefixes \ekvd@empty \ekvd@ifalso
\ekvd@clear@prefixes \ekvd@empty \ekvd@ifalso
20 \def\ekvd@empty{}
```

`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.

```

21 \protected\def\ekvd@clear@prefixes
22   {%
23     \let\ekvd@long\ekvd@empty
24     \let\ekvd@prot\ekvd@empty
25     \let\ekvd@ifalso\@secondoftwo
26     \long\def\ekvd@ifnew##1##2##3{##3}%
27   }
28 \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 `\langle key \rangle = \langle value \rangle` list, the interpretation will be done by `\ekvd@noarg` and `\ekvd@`. The `\langle set \rangle` for which the keys should be defined is stored in `\ekvd@set`.

```

29 \protected\def\ekvdefinekeys#1%
30   {%
31     \def\ekvd@set{#1}%
32     \ekvparse\ekvd@noarg\ekvd@arg
33   }

```

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

`\ekvd@noarg` `\ekvd@noarg` and `\ekvd@arg` store whether there was a value in the `\langle key \rangle = \langle value \rangle` pair.
`\ekvd@arg` `\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.
`\ekvd@handle`

```

34 \protected\def\ekvd@noarg#1%
35   {%
36     \let\ekvd@ifnoarg\@firstoftwo
37     \ekvd@handle{#1}{}%
38   }
39 \protected\def\ekvd@arg
40   {%
41     \let\ekvd@ifnoarg\@secondoftwo
42     \ekvd@handle
43   }
44 \protected\long\def\ekvd@handle#1#2%
45   {%
46     \ekvd@clear@prefixes
47     \edef\ekvd@cur{\detokenize{#1}}%
48     \ekvd@ifspace{#1}%
49     {\ekvd@prefix\ekv@mark#1\ekv@stop{#2}}%
50     \ekvd@err@missing@type
51   }

```

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

`\ekvd@prefix` **`\ekvDEF`** separates prefixes into two groups, the first being prefixes in the T_EX sense
`\ekvd@prefix@` (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).

```

52 \protected\def\ekvd@prefix#1 {\ekv@strip{#1}\ekvd@prefix@\ekv@mark}
53 \protected\def\ekvd@prefix@#1#2\ekv@stop

```

```

54   {%
55   \ekv@ifdefined{ekvd@t@#1}%
56     {\ekv@strip{#2}{\csname ekvd@t@#1\endcsname}}%
57     {%
58     \ekv@ifdefined{ekvd@p@#1}%
59       {\csname ekvd@p@#1\endcsname\ekvd@prefix@after@p{#2}}%
60       {\ekvd@err@undefined@prefix{#1}\@gobble}%
61     }%
62   }

```

(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.

```

63 \protected\def\ekvd@prefix@after@p#1%
64   {%
65   \ekvd@ifspace{#1}%
66     {\ekvd@prefix#1\ekv@stop}%
67     {\ekvd@err@missing@type\@gobble}%
68   }

```

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

`\ekvd@p@long` Define the `@p@` type prefixes, they all just store some information in a temporary macro.
`\ekvd@p@protected`
`\ekvd@p@protect`
`\ekvd@p@also`
`\ekvd@p@new`

```

69 \protected\def\ekvd@p@long{\let\ekvd@long\long}
70 \protected\def\ekvd@p@protected{\let\ekvd@prot\protected}
71 \let\ekvd@p@protect\ekvd@p@protected
72 \protected\def\ekvd@p@also{\let\ekvd@ifalso\@firstoftwo}
73 \protected\def\ekvd@p@new{\let\ekvd@ifnew\ekvd@assert@new}

```

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

2.2.1 Key Types

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

```

74 \protected\def\ekvd@type@set#1#2%
75   {%
76   \ekvd@assert@not@long
77   \ekvd@assert@not@protected
78   \ekvd@ifnew{NoVal}{#1}%
79   {%
80   \ekv@ifempty{#2}%
81     {\ekvd@err@missing@definition}%
82     {%
83     \ekvd@ifalso
84     {%
85     \ekv@expargtwice{\ekvd@add@noval{#1}}%
86     {\ekvchangeset{#2}}%
87     \ekvd@assert@not@protected@also
88     }%
89     {\ekv@expargtwice{\ekvdefNoVal\ekvd@set{#1}}{\ekvchangeset{#2}}}%
90   }%
91   }%

```

```

92 }
93 \protected\def\ekvd@t@set#1#2%
94 {%
95   \ekvd@ifnoarg
96     {\ekvd@type@set{#1}{#1}}%
97     {\ekvd@type@set{#1}{#2}}%
98 }

```

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

`\ekvd@type@noval` Another pretty simple type, `noval` just needs to assert that there is a definition and that `\ekvd@t@noval` long wasn't specified. There are types where the difference in the variants is so small, `\ekvd@t@enoval` 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.

```

99 \protected\long\def\ekvd@type@noval#1#2#3%
100 {%
101   \ekvd@ifnew{NoVal}{#2}%
102   {%
103     \ekvd@assert@arg
104     {%
105       \ekvd@assert@not@long
106       \ekvd@prot#1\ekvd@tmp{#3}%
107       \ekvd@ifalso
108         {\ekv@exparg{\ekvd@add@noval{#2}}\ekvd@tmp{}}%
109         {\ekvletNoVal\ekvd@set{#2}\ekvd@tmp}%
110       }%
111     }%
112   }
113 \protected\def\ekvd@t@noval{\ekvd@type@noval\def}
114 \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` code is simple as well, `ecode` has to use `\edef` on a temporary macro, since `explv` doesn't provide an `\ekvedef`.

```

115 \protected\long\def\ekvd@type@code#1#2#3%
116 {%
117   \ekvd@ifnew{}{#2}%
118   {%
119     \ekvd@assert@arg
120     {%
121       \ekvd@prot\ekvd@long#1\ekvd@tmp##1{#3}%
122       \ekvd@ifalso
123         {\ekv@exparg{\ekvd@add@val{#2}}{\ekvd@tmp{##1}}{}}%
124         {\ekvlet\ekvd@set{#2}\ekvd@tmp}%
125       }%
126     }%
127   }
128 \protected\def\ekvd@t@code{\ekvd@type@code\def}
129 \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@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.

```

130 \protected\long\def\ekvd@type@default#1#2#3#4%
131   {%
132     \ekvd@assert@arg
133     {%
134       \ekvifdefined\ekvd@set{#3}%
135       {%
136         \ekvd@assert@not@new
137         \ekvd@assert@not@long
138         \ekvd@prot\edef\ekvd@tmp
139         {%
140           \unexpanded\expandafter#1%
141           {#2\csname\ekv@name\ekvd@set{#3}\endcsname{#4}}%
142         }%
143         \ekvd@ifalso
144         {\ekv@exparg{\ekvd@add@noval{#3}}\ekvd@tmp{}}%
145         {\ekvletNoVal\ekvd@set{#3}\ekvd@tmp}%
146       }%
147       {\ekvd@err@undefined@key{#3}}%
148     }%
149   }
150 \protected\def\ekvd@t@default{\ekvd@type@default{}}
151 \protected\def\ekvd@t@qdefault{\ekvd@type@default{\expandafter\expandafter}}
152 \let\ekvd@t@odefault\ekvd@t@qdefault
153 \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`.

```

154 \protected\long\def\ekvd@t@edefault#1#2%
155   {%
156     \ekvd@assert@arg
157     {%
158       \ekvifdefined\ekvd@set{#1}%
159       {%
160         \ekvd@assert@not@new
161         \ekvd@assert@not@long
162         \ekvd@prot\edef\ekvd@tmp
163         {\csname\ekv@name\ekvd@set{#1}\endcsname{#2}}%
164         \ekvd@ifalso
165         {\ekv@exparg{\ekvd@add@noval{#1}}\ekvd@tmp{}}%
166         {\ekvletNoVal\ekvd@set{#1}\ekvd@tmp}%
167       }%
168       {\ekvd@err@undefined@key{#1}}%
169     }%
170   }

```

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

```

\ekvd@t@initial
\ekvd@t@oinitial 171 \long\def\ekvd@type@initial#1#2#3#4%
\ekvd@t@finitial 172 {%
\ekvd@t@einitial 173 \ekvd@assert@arg
174 {%
175 \ekvifdefined\ekvd@set{#3}%
176 {%
177 \ekvd@assert@not@new
178 \ekvd@assert@not@also
179 \ekvd@assert@not@long
180 \ekvd@assert@not@protected
181 #1{#2#4}%
182 \cename\ekv@name\ekvd@set{#3}\expandafter\endcename\expandafter
183 {\ekvd@tmp}%
184 }%
185 {\ekvd@err@undefined@key{#3}}%
186 }%
187 }
188 \def\ekvd@t@initial{\ekvd@type@initial{\def\ekvd@tmp}{}}
189 \def\ekvd@t@oinitial{\ekvd@type@initial{\ekv@exparg{\def\ekvd@tmp}{}}}
190 \def\ekvd@t@einitial{\ekvd@type@initial{\edef\ekvd@tmp}{}}
191 \def\ekvd@t@finitial
192 {\ekvd@type@initial{\ekv@exparg{\def\ekvd@tmp}{}}{\romannumeral‘\^^@}}

```

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

```

\ekvd@type@bool The boolean types are a quicker version of a choice that accept true and false, and
\ekvd@t@bool set up the NoVal action to be identical to <key>=true. The true and false actions are
\ekvd@t@gbool always just \letting the macro in #7 to some other macro (e.g., \iftrue).
\ekvd@t@boolTF 193 \protected\def\ekvd@type@bool#1#2#3#4#5%
\ekvd@t@gboolTF 194 {%
\ekvd@t@invbool 195 \ekvd@ifnew{#4}%
\ekvd@t@ginvbool 196 {%
\ekvd@t@invboolTF 197 \ekvd@ifnew{NoVal}{#4}%
\ekvd@t@ginvboolTF 198 {%
199 \ekvd@assert@filledarg{#5}%
200 {%
201 \ekvd@newlet#5#3%
202 \ekvd@type@choice{#4}%
203 \protected\ekvdefNoVal\ekvd@set{#4}{#1\let#5#2}%
204 \protected\expandafter\def
205 \cename\ekvd@choice@name\ekvd@set{#4}{true}\endcename
206 {#1\let#5#2}%
207 \protected\expandafter\def
208 \cename\ekvd@choice@name\ekvd@set{#4}{false}\endcename
209 {#1\let#5#3}%
210 }%
211 }%
212 }%
213 }
214 \protected\def\ekvd@t@bool{\ekvd@type@bool}\iftrue\iffalse}
215 \protected\def\ekvd@t@gbool{\ekvd@type@bool\global\iftrue\iffalse}
216 \protected\def\ekvd@t@boolTF{\ekvd@type@bool}\@firstoftwo\@secondoftwo}
217 \protected\def\ekvd@t@gboolTF{\ekvd@type@bool\global\@firstoftwo\@secondoftwo}

```

```

218 \protected\def\ekvd@t@invbool{\ekvd@type@bool}\iffalse\iftrue}
219 \protected\def\ekvd@t@ginvbool{\ekvd@type@bool\global\iffalse\iftrue}
220 \protected\def\ekvd@t@invboolTF{\ekvd@type@bool}\@secondoftwo\@firstoftwo}
221 \protected\def\ekvd@t@ginvboolTF
222   {\ekvd@type@bool\global\@secondoftwo\@firstoftwo}

```

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

\ekvd@type@boolpair The boolean pair types are essentially the same as the boolean types, but set two macros instead of one.

```

\ekvd@t@boolpair
\ekvd@t@gboolpair
\ekvd@t@boolpairTF
\ekvd@t@gboolpairTF
223 \protected\def\ekvd@type@boolpair#1#2#3#4#5#6%
224   {%
225     \ekvd@ifnew{#4}%
226     {%
227       \ekvd@ifnew{NoVal}{#4}%
228       {%
229         \ekvd@newlet#5#3%
230         \ekvd@newlet#6#2%
231         \ekvd@type@choice{#4}%
232         \protected\ekvdefNoVal\ekvd@set{#4}{#1\let#5#2#1\let#6#3}%
233         \protected\expandafter\def
234           \csname\ekvd@choice@name\ekvd@set{#4}{true}\endcsname
235           {#1\let#5#2#1\let#6#3}%
236         \protected\expandafter\def
237           \csname\ekvd@choice@name\ekvd@set{#4}{false}\endcsname
238           {#1\let#5#3#1\let#6#2}%
239       }%
240     }%
241   }
242 \protected\def\ekvd@t@boolpair#1#2%
243   {\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair}\iftrue\iffalse{#1}#2}}
244 \protected\def\ekvd@t@gboolpair#1#2%
245   {\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair\global\iftrue\iffalse{#1}#2}}
246 \protected\def\ekvd@t@boolpairTF#1#2%
247   {%
248     \ekvd@assert@twoargs{#2}%
249     {\ekvd@type@boolpair}\@firstoftwo\@secondoftwo{#1}#2}%
250   }
251 \protected\def\ekvd@t@gboolpairTF#1#2%
252   {%
253     \ekvd@assert@twoargs{#2}%
254     {\ekvd@type@boolpair\global\@firstoftwo\@secondoftwo{#1}#2}%
255   }

```

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

```

\ekvd@type@data
\ekvd@t@data
\ekvd@t@gdata
\ekvd@t@dataT
\ekvd@t@gdataT
256 \protected\def\ekvd@type@data#1#2#3#4#5#6%
257   {%
258     \ekvd@ifnew{#5}%
259     {%
260       \ekvd@assert@filledarg{#6}%
261       {%
262         \ekvd@newlet#6#1%

```



```

263         \ekvd@ifalso
264         {%
265             \let\ekvd@prot\protected
266             \ekvd@add@val{#5}{\long#2#6###1#3{###1{#4}}}{}%
267         }%
268         {%
269             \protected\ekvd@long\ekvdef\ekvd@set{#5}%
270             {\long#2#6###1#3{###1{#4}}}%
271         }%
272     }%
273 }%
274 }
275 \protected\def\ekvd@t@data
276   {\ekvd@type@data\@secondoftwo\edef{###2}{\unexpanded{##1}}}
277 \protected\def\ekvd@t@edata{\ekvd@type@data\@secondoftwo\edef{###2}{##1}}
278 \protected\def\ekvd@t@gdata
279   {\ekvd@type@data\@secondoftwo\xdef{###2}{\unexpanded{##1}}}
280 \protected\def\ekvd@t@xdata{\ekvd@type@data\@secondoftwo\xdef{###2}{##1}}
281 \protected\def\ekvd@t@dataT{\ekvd@type@data\@gobble\edef{}{\unexpanded{##1}}}
282 \protected\def\ekvd@t@edataT{\ekvd@type@data\@gobble\edef{}{##1}}
283 \protected\def\ekvd@t@gdataT{\ekvd@type@data\@gobble\xdef{}{\unexpanded{##1}}}
284 \protected\def\ekvd@t@xdataT{\ekvd@type@data\@gobble\xdef{}{##1}}

```

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

\ekvd@type@box Set up our boxes. Though we're a generic package we want to be colour safe, so we put an additional grouping level inside the box contents, for the case that someone uses color.
 \ekvd@t@box
 \ekvd@t@gbox \ekvd@newreg is a small wrapper which tests whether the first argument is defined and if not does \csname new#2\endcsname#1.

```

285 \protected\def\ekvd@type@box#1#2#3%
286   {%
287     \ekvd@ifnew{}{#2}%
288     {%
289       \ekvd@assert@filledarg{#3}%
290       {%
291         \ekvd@newreg#3{box}%
292         \ekvd@ifalso
293         {%
294           \let\ekvd@prot\protected
295           \ekvd@add@val{#2}{#1\setbox#3\hbox{\begingroup##1\endgroup}}{}%
296         }%
297         {%
298           \protected\ekvd@long\ekvdef\ekvd@set{#2}%
299           {#1\setbox#3\hbox{\begingroup##1\endgroup}}}%
300         }%
301       }%
302     }%
303   }
304 \protected\def\ekvd@t@box{\ekvd@type@box{}}
305 \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 Similar to box, but set the toks.
 \ekvd@t@toks
 \ekvd@t@gtoks

```

306 \protected\def\ekvd@type@toks#1#2#3%
307   {%
308     \ekvd@ifnew{#2}%
309     {%
310       \ekvd@assert@filledarg{#3}%
311       {%
312         \ekvd@newreg#3{toks}%
313         \ekvd@ifalso
314         {%
315           \let\ekvd@prot\protected
316           \ekvd@add@val{#2}{#1#3{##1}}{}}%
317         }%
318         {\protected\ekvd@long\ekvdef\ekvd@set{#2}{#1#3{##1}}}%
319       }%
320     }%
321   }
322 \protected\def\ekvd@t@toks{\ekvd@type@toks{}}
323 \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 Just like toks, but expand the current contents of the toks register to append the new contents.

```

\ekvd@t@apptoks
\ekvd@t@gapptoks
324 \protected\def\ekvd@type@apptoks#1#2#3%
325   {%
326     \ekvd@ifnew{#2}%
327     {%
328       \ekvd@assert@filledarg{#3}%
329       {%
330         \ekvd@newreg#3{toks}%
331         \ekvd@ifalso
332         {%
333           \let\ekvd@prot\protected
334           \ekvd@add@val{#2}{#1#3\expandafter{\the#3##1}}{}}%
335         }%
336         {%
337           \protected\ekvd@long\ekvdef\ekvd@set{#2}%
338             {#1#3\expandafter{\the#3##1}}}%
339         }%
340       }%
341     }%
342   }
343 \protected\def\ekvd@t@apptoks{\ekvd@type@apptoks{}}
344 \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 The \ekvd@type@reg can handle all the types for which the assignment will just be <register>=(value).

```

\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
345 \protected\def\ekvd@type@reg#1#2#3#4#5#6%
346   {%
347     \ekvd@ifnew{#5}%
348     {%
349       \ekvd@assert@filledarg{#6}%

```

```

350     {%
351     \ekvd@newreg#6{#1}%
352     \ekvd@ifalso
353     {%
354     \let\ekvd@prot\protected
355     \ekvd@add@val{#5}{#2#6=#3##1#4\relax}{}%
356     }%
357     {\protected\ekvd@long\ekvdef\ekvd@set{#5}{#2#6=#3##1#4\relax}}%
358     }%
359   }%
360 }
361 \protected\def\ekvd@t@int{\ekvd@type@reg{count}{-}{-}}
362 \protected\def\ekvd@t@eint{\ekvd@type@reg{count}{-}\numexpr\relax}
363 \protected\def\ekvd@t@gint{\ekvd@type@reg{count}\global{-}{-}}
364 \protected\def\ekvd@t@xint{\ekvd@type@reg{count}\global\numexpr\relax}
365 \protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{-}{-}}
366 \protected\def\ekvd@t@edimen{\ekvd@type@reg{dimen}{-}\dimexpr\relax}
367 \protected\def\ekvd@t@gdimen{\ekvd@type@reg{dimen}\global{-}{-}}
368 \protected\def\ekvd@t@xdimen{\ekvd@type@reg{dimen}\global\dimexpr\relax}
369 \protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{-}{-}}
370 \protected\def\ekvd@t@eskip{\ekvd@type@reg{skip}{-}\glueexpr\relax}
371 \protected\def\ekvd@t@gskip{\ekvd@type@reg{skip}\global{-}{-}}
372 \protected\def\ekvd@t@xskip{\ekvd@type@reg{skip}\global\glueexpr\relax}

```

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

\ekvd@type@store The none-expanding store types use an \edef or \xdef and \unexpanded to be able to also store # easily.

```

\ekvd@t@store
\ekvd@t@gstore
373 \protected\def\ekvd@type@store#1#2#3#4%
374   {%
375   \ekvd@ifnew{-}{#3}%
376   {%
377   \ekvd@assert@filledarg{#4}%
378   {%
379   \ekvd@newlet#4\ekvd@empty
380   \ekvd@ifalso
381   {%
382   \let\ekvd@prot\protected
383   \ekvd@add@val{#3}{#1#4{#2}}{-%
384   }%
385   {\protected\ekvd@long\ekvdef\ekvd@set{#3}{#1#4{#2}}}%
386   }%
387   }%
388 }
389 \protected\def\ekvd@t@store{\ekvd@type@store\edef{\unexpanded{##1}}}
390 \protected\def\ekvd@t@gstore{\ekvd@type@store\xdef{\unexpanded{##1}}}
391 \protected\def\ekvd@t@estore{\ekvd@type@store\edef{##1}}
392 \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 meta sets up things such that another instance of \ekvset will be run on the argument, with the same <set>.

```

\ekvd@type@meta@a
\ekvd@type@meta@b
\ekvd@type@meta@c
\ekvd@t@meta
\ekvd@t@nmeta
393 \protected\long\def\ekvd@type@meta#1#2#3#4#5#6#7%

```

```

394   {%
395     \ekvd@ifnew{#1}{#6}%
396     {%
397       \ekvd@assert@filledarg{#7}%
398       {%
399         \edef\ekvd@tmp{\ekvd@set}%
400         \expandafter\ekvd@type@meta@a\expandafter{\ekvd@tmp}{#7}{#2}%
401         \ekvd@ifalso
402           {\ekv@exparg{#3{#6}}{\ekvd@tmp#4}{#5}}%
403           {\csname ekvlet#1\endcsname\ekvd@set{#6}\ekvd@tmp}%
404       }%
405     }%
406   }
407 \protected\long\def\ekvd@type@meta@a#1#2%
408   {%
409     \expandafter\ekvd@type@meta@b\expandafter{\ekvset{#1}{#2}}%
410   }
411 \protected\def\ekvd@type@meta@b
412   {%
413     \expandafter\ekvd@type@meta@c\expandafter
414   }
415 \protected\long\def\ekvd@type@meta@c#1#2%
416   {%
417     \ekvd@prot\ekvd@long\def\ekvd@tmp#2{#1}%
418   }
419 \protected\def\ekvd@t@meta{\ekvd@type@meta}{##1}\ekvd@add@val{##1}{}}
420 \protected\def\ekvd@t@nmeta
421   {%
422     \ekvd@assert@not@long
423     \ekvd@type@meta{NoVal}{}\ekvd@add@noval{}\ekvd@assert@not@long@also
424   }

```

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

\ekvd@type@smeta smeta is pretty similar to meta, but needs two arguments inside of $\langle value \rangle$, such that the first is the $\langle set \rangle$ for which the sub-\ekvset and the second is the $\langle key \rangle = \langle value \rangle$ list.

```

\ekvd@type@smeta@
\ekvd@t@smeta
\ekvd@t@snmeta
425 \protected\long\def\ekvd@type@smeta#1#2#3#4#5#6#7%
426   {%
427     \ekvd@ifnew{#1}{#6}%
428     {%
429       \ekvd@assert@twoargs{#7}%
430       {%
431         \ekvd@type@meta@a#7{#2}%
432         \ekvd@ifalso
433           {\ekv@exparg{#3{#6}}{\ekvd@tmp#4}{#5}}%
434           {\csname ekvlet#1\endcsname\ekvd@set{#6}\ekvd@tmp}%
435       }%
436     }%
437   }
438 \protected\def\ekvd@t@smeta{\ekvd@type@smeta}{##1}\ekvd@add@val{##1}{}}
439 \protected\def\ekvd@t@snmeta
440   {%
441     \ekvd@assert@not@long
442     \ekvd@type@smeta{NoVal}{}\ekvd@add@noval{}\ekvd@assert@not@long@also
443   }

```

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

`\ekvd@type@choice` The choice type is by far the most complex type, as we have to run a sub-parser on
`\ekvd@populate@choice` the choice-definition list, which should support the `@p@` type prefixes as well (but long
`\ekvd@populate@choice@` will always throw an error, as they are not allowed to be long). `\ekvd@type@choice`
`\ekvd@populate@choice@noarg` will just define the choice-key, the handling of the choices definition will be done by
`\ekvd@choice@prefix` `\ekvd@populate@choice`.
`\ekvd@choice@prefix@`
`\ekvd@choice@p@protected`
`\ekvd@choice@p@protect`
`\ekvd@choice@p@long`
`\ekvd@choice@p@long@`
`\ekvd@t@choice`

```

444 \protected\def\ekvd@type@choice#1%
445   {%
446     \ekvd@assert@not@long
447     \ekvd@prot\edef\ekvd@tmp##1%
448     {\unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#1}{##1}}}%
449     \ekvd@ifalso
450     {%
451       \ekvd@assert@val{#1}%
452       {%
453         \ekvd@if@not@already@choice{#1}%
454         {%
455           \ekv@exparg
456           {%
457             \expandafter\ekvd@add@aux
458             \csname\ekv@name\ekvd@set{#1}\endcsname{##1}{#1}%
459             }%
460             {\ekvd@tmp{##1}}%
461             {\ekvd@long\ekvdef}\ekvd@assert@not@long@also
462           }%
463         }%
464       }%
465     }{\ekvlet\ekvd@set{#1}\ekvd@tmp}%
466   }

```

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

```

467 \protected\def\ekvd@populate@choice
468   {%
469     \ekvparse\ekvd@populate@choice@noarg\ekvd@populate@choice@
470   }
471 \protected\long\def\ekvd@populate@choice@noarg#1%
472   {%
473     \expandafter\ekvd@err@missing@definition@msg\expandafter{\ekvd@cur : #1}%
474   }

```

`\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`.

```

475 \protected\long\def\ekvd@populate@choice@#1#2%
476   {%
477     \ekvd@clear@prefixes
478     \expandafter\ekvd@assert@arg@msg\expandafter{\ekvd@cur : #1}%
479     {%
480       \ekvd@ifspace{#1}%
481       {\ekvd@choice@prefix\ekv@mark#1\ekv@stop}%
482     }%

```

```

483         \expandafter\def
484         \csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname
485         }%
486         {#2}%
487     }%
488 }
489 \protected\def\ekvd@choice@prefix#1
490 {%
491     \ekv@strip{#1}\ekvd@choice@prefix@\ekv@mark
492 }
493 \protected\def\ekvd@choice@prefix@#1#2\ekv@stop
494 {%
495     \ekv@ifdefined{ekvd@choice@p@#1}%
496     {%
497         \csname ekvd@choice@p@#1\endcsname
498         \ekvd@ifspace{#2}%
499         {\ekvd@choice@prefix#2\ekv@stop}%
500         {%
501             \ekvd@prot\expandafter\def
502             \csname
503             \ekv@strip{#2}{\ekvd@choice@name\ekvd@set\ekvd@set@choice}%
504             \endcsname
505         }%
506     }%
507     {\ekvd@err@undefined@prefix{#1}\@gobble}%
508 }
509 \protected\def\ekvd@choice@p@protected{\let\ekvd@prot\protected}
510 \let\ekvd@choice@p@protect\ekvd@choice@p@protected
511 \protected\def\ekvd@choice@invalid@p#1\ekvd@ifspace#2%
512 {%
513     \expandafter\ekvd@choice@invalid@p\expandafter{\ekv@gobble@mark#2}{#1}%
514     \ekvd@ifspace{#2}%
515 }
516 \protected\def\ekvd@choice@invalid@p@#1#2%
517 {%
518     \expandafter\ekvd@err@no@prefix@msg\expandafter{\ekvd@cur : #2 #1}{#2}%
519 }
520 \protected\def\ekvd@choice@p@long{\ekvd@choice@invalid@p{long}}%
521 \protected\def\ekvd@choice@p@also{\ekvd@choice@invalid@p{also}}%
522 \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.

```

523 \protected\long\def\ekvd@t@choice#1#2%
524 {%
525     \ekvd@ifnew{#1}%
526     {%
527         \ekvd@assert@arg
528         {%
529             \ekvd@type@choice{#1}%
530             \def\ekvd@set@choice{#1}%
531             \ekvd@populate@choice{#2}%
532         }%
533     }%

```

```
534 }
```

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

`\ekvd@t@unknown-choice`

```
535 \protected\long\expandafter\def\csname ekvd@t@unknown-choice\endcsname#1#2%
536 {%
537   \ekvd@assert@new@for@name{\ekvd@unknown@choice@name\ekvd@set{#1}}%
538   {%
539     \ekvd@assert@arg
540     {%
541       \ekvd@assert@not@long
542       \ekvd@assert@not@also
543       \ekvd@prot\expandafter
544       \def\csname\ekvd@unknown@choice@name\ekvd@set{#1}\endcsname##1{#2}%
545     }%
546   }%
547 }
```

(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.

```
548 \protected\long\def\ekvd@t@unknown#1#2%
549 {%
550   \ekv@ifdefined{\ekvd@type@unknown@\detokenize{#1}}%
551   {\csname ekvd@type@unknown@\detokenize{#1}\endcsname{#2}}%
552   \ekvd@err@misused@unknown
553 }
```

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

```
554 \protected\long\def\ekvd@type@unknown@noval#1%
555 {%
556   \ekvd@assert@new@for@name{\ekv@name\ekvd@set{ }uN}%
557   {%
558     \ekvd@assert@arg
559     {%
560       \ekvd@assert@not@also
561       \ekvd@assert@not@long
562       \ekvd@prot\ekvdefunknownNoVal\ekvd@set{#1}%
563     }%
564   }%
565 }
```

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.

```
566 \protected\long\def\ekvd@type@unknown@code#1%
567 {%
568   \ekvd@assert@new@for@name{\ekv@name\ekvd@set{ }u}%
569   {%
570     \ekvd@assert@arg
571     {%
```

```

572         \ekvd@assert@not@also
573         \begingroup
574         \def\ekvd@tmp##1##2{#1}%
575         \ekv@exparg
576         {%
577         \endgroup
578         \ekvd@prot\ekvd@long\ekvdefunknown\ekvd@set
579         }%
580         {\ekvd@tmp{##2}{##1}}%
581     }%
582 }%
583 }

```

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

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

```

\ekvd@type@unknown@redirect-code
\ekvd@type@unknown@redirect-noval
584 \protected\edef\ekvd@type@unknown@redirect#1%
585     {%
586     \expandafter\noexpand\csname ekvd@type@unknown@redirect-code\endcsname{#1}%
587     \expandafter\noexpand\csname ekvd@type@unknown@redirect-noval\endcsname{#1}%
588     }
589 \protected\expandafter\def\csname ekvd@type@unknown@redirect-code\endcsname#1%
590     {%
591     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}}u}%
592     {%
593     \ekvd@assert@arg
594     {%
595     \ekvd@assert@not@also
596     \ekvd@assert@not@protected
597     \expandafter\ekvredirectunknown\expandafter{\ekvd@set}{#1}%
598     }%
599     }%
600     }
601 \protected\expandafter\def\csname ekvd@type@unknown@redirect-noval\endcsname#1%
602     {%
603     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}}uN}%
604     {%
605     \ekvd@assert@arg
606     {%
607     \ekvd@assert@not@also
608     \ekvd@assert@not@protected
609     \ekvd@assert@not@long
610     \expandafter\ekvredirectunknownNoVal\expandafter{\ekvd@set}{#1}%
611     }%
612     }%
613     }

```

(End definition for \ekvd@type@unknown@redirect, \ekvd@type@unknown@redirect-code, and \ekvd@type@unknown@redirect-noval.)

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` 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.

```

614 \def\ekvd@h@choice#1%
615   {%
616     \expandafter\ekvd@h@choice@
617     \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname
618     {#1}%
619   }
620 \def\ekvd@h@choice@#1#2%
621   {%
622     \ifx#1\relax
623       \ekvd@err@choice@invalid{#2}%
624       \expandafter\@gobble
625     \fi
626     #1%
627   }

```

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

2.2.3 Handling also

```

\ekvd@add@val
\ekvd@add@noval 628 \protected\long\def\ekvd@add@val#1#2#3%
\ekvd@add@aux    629   {%
\ekvd@add@aux@  630     \ekvd@assert@val{#1}%
631     {%
632       \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}\endcsname-{{#1}}%
633       {#1}{#2}{\ekvd@long\ekvdef}{#3}%
634     }%
635   }
636 \protected\long\def\ekvd@add@noval#1#2#3%
637   {%
638     \ekvd@assert@noval{#1}%
639     {%
640       \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}N\endcsname{}%
641       {#1}{#2}\ekvdefNoVal{#3}%
642     }%
643   }
644 \protected\long\def\ekvd@add@aux#1#2%
645   {%
646     \ekvd@extract@prefixes#1%
647     \expandafter\ekvd@add@aux@\expandafter{#1#2}%
648   }
649 \protected\long\def\ekvd@add@aux@#1#2#3#4#5%
650   {%
651     #5%
652     \ekvd@prot#4\ekvd@set{#2}{#1#3}%
653   }

```

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

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

```

654 \protected\def\ekvd@extract@prefixes#1%

```

```

655   {%
656   \expandafter\ekvd@extract@prefixes@\meaning#1\ekvd@stop
657   }

```

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.

```

658 \protected\def\ekvd@extract@prefixes@#1#2#3%
659   {%
660   \protected\def\ekvd@extract@prefixes@##1#1##2\ekvd@stop
661   {%
662   \ekvd@extract@prefixes@long
663   ##1\ekvd@mark@\@firstofone#2\ekvd@mark@\@gobble\ekvd@stop
664   {\let\ekvd@long\long}%
665   \ekvd@extract@prefixes@prot
666   ##1\ekvd@mark@\@firstofone#3\ekvd@mark@\@gobble\ekvd@stop
667   {\let\ekvd@prot\protected}%
668   }%
669   \protected\def\ekvd@extract@prefixes@long##1#2##2\ekvd@mark###3###4\ekvd@stop
670   {##3}%
671   \protected\def\ekvd@extract@prefixes@prot##1#3##2\ekvd@mark###3###4\ekvd@stop
672   {##3}%
673   }

```

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.

```

674 \begingroup
675 \edef\ekvd@tmp
676   {%
677   \endgroup
678   \ekvd@extract@prefixes@
679   {\detokenize{macro:}}%
680   {\string\long}%
681   {\string\protected}%
682   }
683 \ekvd@tmp

```

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

2.2.4 Tests

\ekvd@newlet 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>.

```

684 \protected\def\ekvd@newlet#1#2%
685   {%
686   \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\let#1#2}%
687   }
688 \protected\def\ekvd@newreg#1#2%
689   {%
690   \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\csname new#2\endcsname#1}%
691   }

```

(End definition for \ekvd@newlet and \ekvd@newreg.)

`\ekvd@assert@twoargs` A test for exactly two tokens can be reduced for an empty-test after gobbling two tokens,
`\ekvd@ifnottwoargs` in the case that there are fewer tokens than two in the argument, only macros will be
`\ekvd@ifempty@gtwo` gobbled that are needed for the true branch, which doesn't hurt, and if there are more
this will not be empty.

```
692 \long\def\ekvd@assert@twoargs#1%
693   {%
694     \ekvd@ifnottwoargs{#1}{\ekvd@err@missing@definition}%
695   }
696 \long\def\ekvd@ifnottwoargs#1%
697   {%
698     \ekvd@ifempty@gtwo#1\ekv@ifempty@B
699     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
700   }
701 \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` Assert that a given key is defined as a value taking key or a NoVal key with the correct
`\ekvd@assert@val@` argument structure, respectively.

```
702 \protected\def\ekvd@assert@val#1%
703   {%
704     \ekvifdefined\ekvd@set{#1}%
705     {\expandafter\ekvd@assert@val@\csname\ekv@name\ekvd@set{#1}\endcsname}%
706     {%
707       \ekvifdefinedNoVal\ekvd@set{#1}%
708       \ekvd@err@add@val@on@noval
709       {\ekvd@err@undefined@key{#1}}%
710       \@gobble
711     }%
712   }
713 \protected\def\ekvd@assert@val@#1%
714   {%
715     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
716     \unless\ifx\ekvd@extracted@args\ekvd@one@arg@string
717     \ekvd@err@unsupported@arg
718     \fi
719     \@firstofone
720   }%
721 \protected\def\ekvd@assert@noval#1%
722   {%
723     \ekvifdefinedNoVal\ekvd@set{#1}%
724     {\expandafter\ekvd@assert@noval@\csname\ekv@name\ekvd@set{#1}N\endcsname}%
725     {%
726       \ekvifdefined\ekvd@set{#1}%
727       \ekvd@err@add@noval@on@val
728       {\ekvd@err@undefined@key{#1}}%
729       \@gobble
730     }%
731   }
732 \protected\def\ekvd@assert@noval@#1%
733   {%
734     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
735     \unless\ifx\ekvd@extracted@args\ekvd@empty
736     \ekvd@err@unsupported@arg
```

```

737     \fi
738     \@firstofone
739   }
740 \protected\def\ekvd@extract@args#1%
741   {%
742     \protected\def\ekvd@extract@args##1##2->##3\ekvd@stop
743     {\def\ekvd@extracted@args{##2}}%
744   }
745 \expandafter\ekvd@extract@args\expandafter{\detokenize{macro:}}
746 \edef\ekvd@one@arg@string{\string#1}

```

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

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

```

\ekvd@ifnoarg 747 \def\ekvd@assert@arg{\ekvd@ifnoarg\ekvd@err@missing@definition}
748 \long\def\ekvd@assert@arg@msg#1%
749   {%
750     \ekvd@ifnoarg{\ekvd@err@missing@definition@msg{#1}}%
751   }

```

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

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

```

752 \long\def\ekvd@assert@filledarg#1%
753   {%
754     \ekvd@ifnoarg@or@empty{#1}\ekvd@err@missing@definition
755   }
756 \long\def\ekvd@ifnoarg@or@empty#1%
757   {%
758     \ekvd@ifnoarg
759     \@firstoftwo
760     {\ekv@ifempty{#1}}%
761   }

```

(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@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.

```

762 \def\ekvd@assert@not@long{\ifx\ekvd@long\long\ekvd@err@no@prefix{long}\fi}
763 \def\ekvd@assert@not@protected
764   {\ifx\ekvd@prot\protected\ekvd@err@no@prefix{protected}\fi}
765 \def\ekvd@assert@not@also{\ekvd@ifalso{\ekvd@err@no@prefix{also}}{}}
766 \def\ekvd@assert@not@long@also
767   {\ifx\ekvd@long\long\ekvd@err@no@prefix@also{long}\fi}
768 \def\ekvd@assert@not@protected@also
769   {\ifx\ekvd@prot\protected\ekvd@err@no@prefix@also{protected}\fi}
770 \def\ekvd@assert@new#1#2%
771   {\csname ekvifdefined#1\endcsname\ekvd@set{#2}{\ekvd@err@not@new}}
772 \def\ekvd@assert@not@new
773   {\ifx\ekvd@ifnew\ekvd@assert@new\ekvd@err@no@prefix{new}\fi}
774 \def\ekvd@assert@new@for@name#1%
775   {%

```

```

776 \ifx\ekvd@ifnew\ekvd@assert@new
777 \ekv@fi@firstoftwo
778 \fi
779 \@secondoftwo
780 {\ekv@ifdefined{#1}\ekvd@err@not@new}%
781 \@firstofone
782 }

```

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

`\ekvd@if@not@already@choice` 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.

```

783 \protected\def\ekvd@if@not@already@choice#1%
784 {%
785 \expandafter\ekvd@if@not@already@choice@a
786 \csname\ekv@name\ekvd@set{#1}\endcsname
787 { }\ekvd@h@choice\ekvd@stop
788 }
789 \protected\def\ekvd@if@not@already@choice@a
790 {%
791 \expandafter\ekvd@if@not@already@choice@b
792 }
793 \long\protected\def\ekvd@if@not@already@choice@b#1\ekvd@h@choice#2\ekvd@stop
794 {%
795 \ekv@ifempty{#2}\@firstofone\@gobble
796 }

```

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

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

```

797 \long\def\ekvd@ifspace#1%
798 {%
799 \ekvd@ifspace@#1 \ekv@ifempty@B
800 \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
801 }
802 \long\def\ekvd@ifspace@#1 % keep this space
803 {%
804 \ekv@ifempty@\ekv@ifempty@A
805 }

```

(End definition for `\ekvd@ifspace` and `\ekvd@ifspace@`.)

2.2.5 Messages

Most messages of `expKVDEF` are not expandable, since they only appear during key-definition, which is not expandable anyway.

`\ekvd@errm` The non-expandable error messages are boring, so here they are:

```

\ekvd@err@missing@definition 806 \protected\def\ekvd@errm#1{\errmessage{expkv-def Error: #1}}
\ekvd@err@missing@definition@msg 807 \protected\def\ekvd@err@missing@definition
\ekvd@err@missing@type 808 {\ekvd@errm{Missing definition for key ‘\ekvd@cur’}}
\ekvd@err@undefined@prefix 809 \protected\def\ekvd@err@missing@definition@msg#1%
\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

```

```

810   {\ekvd@errm{Missing definition for key ‘\unexpanded{#1}’}}
811 \protected\def\ekvd@err@missing@type
812   {\ekvd@errm{Missing type prefix for key ‘\ekvd@cur’}}
813 \protected\def\ekvd@err@undefined@prefix#1%
814   {%
815     \ekvd@errm
816       {Undefined prefix ‘\unexpanded{#1}’ found while processing ‘\ekvd@cur’}%
817   }
818 \protected\def\ekvd@err@undefined@key#1%
819   {%
820     \ekvd@errm
821       {Undefined key ‘\unexpanded{#1}’ found while processing ‘\ekvd@cur’}%
822   }
823 \protected\def\ekvd@err@no@prefix#1%
824   {\ekvd@errm{prefix ‘#1’ not accepted in ‘\ekvd@cur’}}
825 \protected\def\ekvd@err@no@prefix@msg#1#2%
826   {\ekvd@errm{prefix ‘#2’ not accepted in ‘\unexpanded{#1}’}}
827 \protected\def\ekvd@err@no@prefix@also#1%
828   {\ekvd@errm{‘\ekvd@cur’ not allowed with a ‘#1’ key}}
829 \protected\def\ekvd@err@add@val@on@noval
830   {\ekvd@errm{‘\ekvd@cur’ not allowed with a NoVal key}}
831 \protected\def\ekvd@err@add@noval@on@val
832   {\ekvd@errm{‘\ekvd@cur’ not allowed with a value taking key}}
833 \protected\def\ekvd@err@unsupported@arg\fi\@firstofone#1%
834   {%
835     \fi
836     \ekvd@errm
837       {%
838         Existing key-macro has the unsupported argument string
839         ‘\ekvd@extracted@args’ for key ‘\ekvd@cur’%
840       }%
841   }
842 \protected\def\ekvd@err@not@new
843   {\ekvd@errm{The key for ‘\ekvd@cur’ is already defined}}
844 \protected\long\def\ekvd@err@misused@unknown
845   {\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 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 \ekvd@choice@name catcode 8 here, which should suffice in most cases to allow for a correct separation.

```

\ekvd@unknown@choice@name
846 \def\ekvd@err@choice@invalid#1%
847   {%
848     \ekvd@err@choice@invalid@#1\ekv@stop
849   }
850 \begingroup
851 \catcode40=8
852 \catcode41=8
853 \@firstofone{\endgroup
854 \def\ekvd@choice@name#1#2#3%
855   {%
856     ekvd#1(#2)#3%
857   }

```

```

858 \def\ekvd@unknown@choice@name#1#2%
859   {%
860     ekvd:u:#1(#2)%
861   }
862 \def\ekvd@err@choice@invalid@ ekvd#1(#2)#3\ekv@stop%
863   {%
864     \ekv@ifdefined{\ekvd@unknown@choice@name{#1}{#2}}%
865     {\csname\ekvd@unknown@choice@name{#1}{#2}\endcsname{#3}}%
866     {\ekvd@err{invalid choice ‘#3’ for ‘#2’ in set ‘#1’}}%
867   }
868 }

```

(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.

```

869 \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 `@`.

```

870 \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.

A		estore	5
also	3	F	
apptoks	6	fdefault	4
B		finitial	4
bool	4	G	
boolpair	5	gapptoks	6
boolpairTF	5	gbool	4
boolTF	4	gboolpair	5
box	6	gboolpairTF	5
C		gboolTF	4
choice	6	gbox	6
code	4	gdata	5
D		gdataT	5
data	5	gdimen	5
dataT	5	gint	5
default	4	ginvbool	5
dimen	5	ginvboolTF	5
E		gskip	5
ecode	4	gstore	5
edata	5	gtoks	6
edataT	5	I	
edefault	4	initial	4
edimen	5	int	5
einitial	4	invbool	5
eint	5	invboolTF	5
\ekvchangeset	86, 89	L	
\ekvdDate	2, 5, 9, <u>15</u>	long	3
\ekvdef 269, 298, 318, 337, 357, 385, 461, 633		M	
\ekvdefinekeys	2, <u>29</u>	meta	6
\ekvdefNoVal	89, 203, 232, 641	N	
\ekvdefunknown	578	new	3
\ekvdefunknownNoVal	562	nmeta	6
\ekvdVersion	2, 5, 9, <u>15</u>	\noexpand	586, 587
\ekvverr	869	noval	4
\ekvifdefined	134, 158, 175, 704, 726	O	
\ekvifdefinedNoVal	707, 723	odefault	4
\ekvlet	124, 465	oinitial	4
\ekvletNoVal	109, 145, 166	P	
\ekvparse	32, 469	protect	3
\ekvredirectunknown	597	protected	3
\ekvredirectunknownNoVal	610	Q	
\ekvset	409	qdefault	4
enoval	4		
eskip	5		

S

set 6
 skip 5
 smeta 6
 snmeta 6
 store 5

T

TEX and L^AT_EX 2_ε commands:

\ekv@exparg 108, 123, 144,
 165, 189, 192, 402, 433, 455, 575, 869
 \ekv@expargtwice 85, 89
 \ekv@fi@firstoftwo 777
 \ekv@fi@gobble 686, 690
 \ekv@gobble@mark 513
 \ekv@ifdefined 55, 58, 495, 550, 780, 864
 \ekv@ifempty 80, 760, 795
 \ekv@ifempty@ 701, 804
 \ekv@ifempty@A 699, 701, 800, 804
 \ekv@ifempty@B 698, 699, 799, 800
 \ekv@ifempty@false 699, 800
 \ekv@mark 49, 52, 481, 491
 \ekv@name .. 141, 163, 182, 458, 556,
 568, 591, 603, 632, 640, 705, 724, 786
 \ekv@stop
 ... 49, 53, 66, 481, 493, 499, 848, 862
 \ekv@strip 52, 56, 491, 503
 \ekvd@add@aux 457, 628
 \ekvd@add@aux@ 628
 \ekvd@add@noval
 85, 108, 144, 165, 423, 442, 628
 \ekvd@add@val 123, 266,
 295, 316, 334, 355, 383, 419, 438, 628
 \ekvd@arg 32, 34
 \ekvd@assert@arg 103, 119, 132, 156,
 173, 527, 539, 558, 570, 593, 605, 747
 \ekvd@assert@arg@msg 478, 747
 \ekvd@assert@filledarg 199,
 260, 289, 310, 328, 349, 377, 397, 752
 \ekvd@assert@new 73, 762
 \ekvd@assert@new@for@name
 537, 556, 568, 591, 603, 774
 \ekvd@assert@not@also
 178, 542, 560, 572, 595, 607, 765
 \ekvd@assert@not@also@UUUUU\ekvd@assert@not@long@also
 762
 \ekvd@assert@not@long
 76, 105, 137, 161,
 179, 422, 441, 446, 541, 561, 609, 762
 \ekvd@assert@not@long@also
 423, 442, 461
 \ekvd@assert@not@new 136, 160, 177, 762

\ekvd@assert@not@protected
 77, 180, 596, 608, 762
 \ekvd@assert@not@protected@also
 87, 762
 \ekvd@assert@noval 638, 702
 \ekvd@assert@noval@ 702
 \ekvd@assert@twoargs
 243, 245, 248, 253, 429, 692
 \ekvd@assert@val 451, 630, 702
 \ekvd@assert@val@ 702
 \ekvd@choice@invalid@p
 511, 520, 521, 522
 \ekvd@choice@invalid@p@ ... 513, 516
 \ekvd@choice@name
 205, 208, 234, 237, 448, 484, 503, 846
 \ekvd@choice@p@also 521
 \ekvd@choice@p@long 444
 \ekvd@choice@p@long@ 444
 \ekvd@choice@p@new 522
 \ekvd@choice@p@protect 444
 \ekvd@choice@p@protected 444
 \ekvd@choice@prefix 444
 \ekvd@choice@prefix@ 444
 \ekvd@clear@prefixes 20, 46, 477
 \ekvd@cur
 .. 47, 473, 478, 518, 808, 812, 816,
 821, 824, 828, 830, 832, 839, 843, 845
 \ekvd@empty 20, 379, 735
 \ekvd@err 866, 869
 \ekvd@err@add@noval@on@val . 727, 806
 \ekvd@err@add@val@on@noval . 708, 806
 \ekvd@err@choice@invalid .. 623, 846
 \ekvd@err@choice@invalid@ 846
 \ekvd@err@missing@definition ...
 81, 694, 747, 754, 806
 \ekvd@err@missing@definition@msg
 473, 750, 806
 \ekvd@err@missing@type ... 50, 67, 806
 \ekvd@err@misused@unknown .. 552, 844
 \ekvd@err@no@prefix
 762, 764, 765, 773, 806
 \ekvd@err@no@prefix@also 767, 769, 806
 \ekvd@err@no@prefix@msg ... 518, 806
 \ekvd@err@not@new 771, 780, 806
 \ekvd@err@not@new@UUUUU\ekvd@err@not@new@undefined@key
 147, 168, 185, 709, 728, 806
 \ekvd@err@undefined@prefix
 60, 507, 806
 \ekvd@err@unsupported@arg
 717, 736, 806
 \ekvd@errm 806
 \ekvd@extract@args 702

<code>\ekvd@extract@prefixes</code>	646 , 654	<code>\ekvd@set</code>	31 , 89 , 109 ,
<code>\ekvd@extract@prefixes@</code>	654		124 , 134 , 141 , 145 , 158 , 163 , 166 ,
<code>\ekvd@extract@prefixes@long</code>	654		175 , 182 , 203 , 205 , 208 , 232 , 234 ,
<code>\ekvd@extract@prefixes@prot</code>	654		237 , 269 , 298 , 318 , 337 , 357 , 385 ,
<code>\ekvd@extracted@args</code>	702 , 839		399 , 403 , 434 , 448 , 458 , 465 , 484 ,
<code>\ekvd@h@choice</code>	448 , 614 , 787 , 793		503 , 537 , 544 , 556 , 562 , 568 , 578 ,
<code>\ekvd@h@choice@</code>	614		591 , 597 , 603 , 610 , 632 , 640 , 652 ,
<code>\ekvd@handle</code>	34		704 , 705 , 707 , 723 , 724 , 726 , 771 , 786
<code>\ekvd@if@not@already@choice</code> 453 , 783		<code>\ekvd@set@choice</code>	484 , 503 , 530
<code>\ekvd@if@not@already@choice@a</code> . .	783	<code>\ekvd@stop</code>	656 , 660 , 663 ,
<code>\ekvd@if@not@already@choice@b</code> . .	783		666 , 669 , 671 , 715 , 734 , 742 , 787 , 793
<code>\ekvd@ifalso</code>	20 ,	<code>\ekvd@t@apptoks</code>	324
	72 , 83 , 107 , 122 , 143 , 164 , 263 , 292 ,	<code>\ekvd@t@bool</code>	193
	313 , 331 , 352 , 380 , 401 , 432 , 449 , 765	<code>\ekvd@t@boolpair</code>	223
<code>\ekvd@ifempty@gtwo</code>	692	<code>\ekvd@t@boolpairTF</code>	223
<code>\ekvd@ifnew</code>	26 , 73 , 78 , 101 , 117 ,	<code>\ekvd@t@boolTF</code>	193
	195 , 197 , 225 , 227 , 258 , 287 , 308 ,	<code>\ekvd@t@box</code>	285
	326 , 347 , 375 , 395 , 427 , 525 , 773 , 776	<code>\ekvd@t@choice</code>	444
<code>\ekvd@ifnoarg</code>	36 , 41 , 95 , 747 , 758	<code>\ekvd@t@code</code>	115
<code>\ekvd@ifnoarg@or@empty</code>	752	<code>\ekvd@t@data</code>	256
<code>\ekvd@ifnottwoargs</code>	692	<code>\ekvd@t@dataT</code>	256
<code>\ekvd@ifspace</code>		<code>\ekvd@t@default</code>	130
	48 , 65 , 480 , 498 , 511 , 514 , 797	<code>\ekvd@t@dimen</code>	345
<code>\ekvd@ifspace@</code>	797	<code>\ekvd@t@ecode</code>	115
<code>\ekvd@long</code>	20 ,	<code>\ekvd@t@edata</code>	277
	69 , 121 , 269 , 298 , 318 , 337 , 357 ,	<code>\ekvd@t@edataT</code>	282
	385 , 417 , 461 , 578 , 633 , 664 , 762 , 767	<code>\ekvd@t@edefault</code>	154
<code>\ekvd@mark</code>	663 , 666 , 669 , 671	<code>\ekvd@t@edimen</code>	345
<code>\ekvd@newlet</code> 201 , 229 , 230 , 262 , 379 , 684		<code>\ekvd@t@einitial</code>	171
<code>\ekvd@newreg</code>	291 , 312 , 330 , 351 , 684	<code>\ekvd@t@eint</code>	345
<code>\ekvd@noarg</code>	32 , 34	<code>\ekvd@t@enoval</code>	99
<code>\ekvd@one@arg@string</code>	702	<code>\ekvd@t@eskip</code>	345
<code>\ekvd@p@also</code>	69	<code>\ekvd@t@estore</code>	391
<code>\ekvd@p@long</code>	69	<code>\ekvd@t@fdefault</code>	130
<code>\ekvd@p@new</code>	69	<code>\ekvd@t@finitial</code>	171
<code>\ekvd@p@protect</code>	69	<code>\ekvd@t@gapptoks</code>	324
<code>\ekvd@p@protected</code>	69	<code>\ekvd@t@gbool</code>	193
<code>\ekvd@populate@choice</code>	444	<code>\ekvd@t@gboolpair</code>	223
<code>\ekvd@populate@choice@</code>	444	<code>\ekvd@t@gboolpairTF</code>	223
<code>\ekvd@populate@choice@noarg</code>	444	<code>\ekvd@t@gboolTF</code>	193
<code>\ekvd@prefix</code>	49 , 52 , 66	<code>\ekvd@t@gbox</code>	285
<code>\ekvd@prefix@</code>	52	<code>\ekvd@t@gdata</code>	256
<code>\ekvd@prefix@after@p</code>	59 , 63	<code>\ekvd@t@gdataT</code>	256
<code>\ekvd@prot</code>		<code>\ekvd@t@gdimen</code>	345
	20 , 70 , 106 , 121 , 138 , 162 , 265 ,	<code>\ekvd@t@gint</code>	345
	294 , 315 , 333 , 382 , 417 , 447 , 501 ,	<code>\ekvd@t@ginbool</code>	193
	509 , 543 , 562 , 578 , 652 , 667 , 764 , 769	<code>\ekvd@t@ginboolTF</code>	193

