

expkv|DEF

a key-defining frontend for expkv

Jonathan P. Spratte*

2020-10-12 v0.6

Abstract

expkv|DEF provides a small $\langle key \rangle = \langle value \rangle$ 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

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	7
1.5	License	8
2	Implementation	9
2.1	The L ^A T _E X Package	9
2.2	The Generic Code	9
2.2.1	Key Types	11
2.2.2	Key Type Helpers	22
2.2.3	Handling also	22
2.2.4	Tests	24
2.2.5	Messages	27
	Index	29

*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 `expkvDEF`, 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, `expkvDEF` works using prefixes instead of suffixes (e.g., `.t1_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. `expkvDEF`'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](#).

`expkvDEF` 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 edefault	default $\langle key \rangle = \{ \langle definition \rangle \}$	new also protected long
initial oinitial 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
invbool ginvbool invboolTF ginvboolTF	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 $\langle key \rangle$. 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.

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 \TeX -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.

These are inverse boolean keys, they behave like bool and friends but set the opposite meaning to the macro $\langle cs \rangle$ in each case. So if $\text{key} = \text{true}$ is used invbool will set $\langle cs \rangle$ to \backslashiffalse and vice versa.

boolpair gboolpair boolpairTF gboolpairTF	boolpair $\langle key \rangle = \langle cs_1 \rangle \langle cs_2 \rangle$	new also protected long
	The boolpair key type behaves like both bool and invbool , the $\langle cs_1 \rangle$ will be set to the meaning according to the rules of bool , 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 $\backslash foo$. This will define $\langle key \rangle$ to store $\langle value \rangle$ inside of the control sequence. If $\langle cs \rangle$ isn't yet defined it will be initialised as empty. The variants behave similarly to their $\backslash def$, $\backslash edef$, $\backslash gdef$, and $\backslash xdef$ counterparts, but store and gstore will allow you to store macro parameters inside of them by using $\backslash unexpanded$.	
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 $\backslash foo$. This will define $\langle key \rangle$ to store $\langle value \rangle$ inside of the control sequence. But unlike the store type, the macro $\langle cs \rangle$ will be a switch at the same time, it'll take two arguments and if $\langle key \rangle$ was used 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 $\backslash @secondoftwo$). The idea is that with this type you can define a key which should be typeset formatted. The edata and xdata variants will fully expand $\langle value \rangle$, the gdata and xdata variants will store $\langle value \rangle$ inside $\langle cs \rangle$ globally. The p-prefixes will only affect the key-macro, $\langle cs \rangle$ will always be expandable and long .	
dataT edataT gdataT xdataT	dataT $\langle key \rangle = \langle cs \rangle$	new also protected long
	Just like data , but instead of $\langle cs \rangle$ grabbing two arguments it'll only grab one, so by default it'll behave like $\backslash @gobble$, and if a $\langle value \rangle$ was given to $\langle key \rangle$ the $\langle cs \rangle$ will behave like $\backslash @firstofone$ 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 $\backslash foo$. An int key will be a TeX-count register. If $\langle cs \rangle$ isn't defined yet, $\backslash newcount$ will be used to initialise it. The eint and xint versions will use $\backslash numexpr$ to allow basic computations in their $\langle value \rangle$. The gint and xint 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 $\backslash foo$. This is just like int but uses a dimen register, $\backslash newdimen$ and $\backslash dimexpr$ 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 $\backslash foo$. This is just like int but uses a skip register, $\backslash newskip$ and $\backslash glueexpr$ instead.	
toks gtoks apptoks gapptoks	toks $\langle key \rangle = \langle cs \rangle$	new also protected long
	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 <key> = <cs>` new also protected long

gbox 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. `expkvDEF` 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.

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

Copyright © 2020 Jonathan P. Spratte

This work may be distributed and/or modified under the conditions of the L^AT_EX 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{0.6}
16 \def\ekvdDate{2020-10-12}
```

(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@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
`\ekvd@handle` grabbing routine, else we throw an error and ignore the key.

```

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@expB@unbraceA{\ekv@expB@unbraceA{\ekvd@add@noval{#1}}}%
86   {\ekvchangeset{#2}}%
87   \ekvd@assert@not@protected@also
88   }%
89   {%
90   \ekv@expB@unbraceA
91   {\ekv@expB@unbraceA{\ekvdefNoVal\ekvd@set{#1}}}%

```

```

92         {\ekvchangeset{#2}}%
93     }%
94 }%
95 }%
96 }
97 \protected\def\ekvd@t@set#1#2%
98 {%
99     \ekvd@ifnoarg
100     {\ekvd@type@set{#1}{#1}}%
101     {\ekvd@type@set{#1}{#2}}%
102 }

```

(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.

```

103 \protected\long\def\ekvd@type@noval#1#2#3%
104 {%
105     \ekvd@ifnew{NoVal}{#2}%
106     {%
107         \ekvd@assert@arg
108         {%
109             \ekvd@assert@not@long
110             \ekvd@prot#1\ekvd@tmp{#3}%
111             \ekvd@ifalso
112             {\ekv@expB@unbraceA{\ekvd@add@noval{#2}}\ekvd@tmp{}}%
113             {\ekvletNoVal\ekvd@set{#2}\ekvd@tmp}%
114         }%
115     }%
116 }
117 \protected\def\ekvd@t@noval{\ekvd@type@noval\def}
118 \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 `expkv` doesn't
\ekvd@t@code provide an \ekvedef.
\ekvd@t@ecode

```

119 \protected\long\def\ekvd@type@code#1#2#3%
120 {%
121     \ekvd@ifnew{}{#2}%
122     {%
123         \ekvd@assert@arg
124         {%
125             \ekvd@prot\ekvd@long#1\ekvd@tmp##1{#3}%
126             \ekvd@ifalso
127             {\ekv@expB@unbraceA{\ekvd@add@val{#2}}{\ekvd@tmp{##1}}{}}%
128             {\ekvlet\ekvd@set{#2}\ekvd@tmp}%
129         }%
130     }%
131 }
132 \protected\def\ekvd@t@code{\ekvd@type@code\def}
133 \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.

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

(End definition for `\ekvd@type@default`, `\ekvd@t@default`, and `\ekvd@t@qdefault`.)

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

```
156 \protected\long\def\ekvd@t@edefault#1#2%
157   {%
158     \ekvd@assert@arg
159     {%
160       \ekvifdefined\ekvd@set{#1}%
161       {%
162         \ekvd@assert@not@new
163         \ekvd@assert@not@long
164         \ekvd@prot\edef\ekvd@tmp
165         {\csname\ekv@name\ekvd@set{#1}\endcsname{#2}}%
166         \ekvd@ifalso
167         {\ekv@expB@unbraceA{\ekvd@add@noval{#1}}\ekvd@tmp{}}%
168         {\ekvletNoVal\ekvd@set{#1}\ekvd@tmp}%
169       }%
170       {\ekvd@err@undefined@key{#1}}%
171     }%
172   }
```

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

```

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

```

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

```

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

```

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 306 \protected\def\ekvd@type@toks#1#2#3%
\ekvd@t@gtoks 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@expB@unbraceA{#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@expB@unbraceA{#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 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`.

```

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@expB@unbraceA
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@prefix0#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     \ifx\ekvd@ifnew\ekvd@assert@new
538       \ekv@fi@firstoftwo
539     \fi
540     \@secondoftwo
541     {\ekv@ifdefined{\ekvd@unknown@choice@name\ekvd@set{#1}}\ekvd@err@not@new}%
542     \@firstofone
543     {%
544       \ekvd@assert@arg
545       {%
546         \ekvd@assert@not@long
547         \ekvd@assert@not@also
548         \ekvd@prot\expandafter
549         \def\csname\ekvd@unknown@choice@name\ekvd@set{#1}\endcsname##1{#2}%
550       }%
551     }%
552 }
```

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

2.2.2 Key Type Helpers

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

`\ekvd@h@choice`
`\ekvd@h@choice@`

The choice helper will just test whether the given choice was defined, if not throw an error expandably, else call the macro which stores the code for this choice.

```
553 \def\ekvd@h@choice#1%
554   {%
555     \expandafter\ekvd@h@choice@
556     \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname
557     {#1}%
558   }
559 \def\ekvd@h@choice@#1#2%
560   {%
561     \ifx#1\relax
562       \ekvd@err@choice@invalid{#2}%
563       \expandafter\@gobble
564     \fi
565     #1%
566   }
```

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

2.2.3 Handling also

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

```
567 \protected\long\def\ekvd@add@val#1#2#3%
568   {%
569     \ekvd@assert@val{#1}%
570     {%
571       \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}\endcsname{{#1}}%
```

```

572         {#1}{#2}{\ekvd@long\ekvdef}{#3}%
573     }%
574 }
575 \protected\long\def\ekvd@add@noval#1#2#3%
576 {%
577     \ekvd@assert@noval{#1}%
578     {%
579         \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}N\endcsname}%
580         {#1}{#2}\ekvdefNoVal{#3}%
581     }%
582 }
583 \protected\long\def\ekvd@add@aux#1#2%
584 {%
585     \ekvd@extract@prefixes#1%
586     \expandafter\ekvd@add@aux@\expandafter{#1#2}%
587 }
588 \protected\long\def\ekvd@add@aux@#1#2#3#4#5%
589 {%
590     #5%
591     \ekvd@prot#4\ekvd@set{#2}{#1#3}%
592 }

```

(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.

```

593 \protected\def\ekvd@extract@prefixes#1%
594 {%
595     \expandafter\ekvd@extract@prefixes@\meaning#1\ekvd@stop
596 }

```

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.

```

597 \protected\def\ekvd@extract@prefixes@#1#2#3%
598 {%
599     \protected\def\ekvd@extract@prefixes@##1#1#2\ekvd@stop
600     {%
601         \ekvd@extract@prefixes@long
602         ##1\ekvd@mark\@firstofone#2\ekvd@mark\@gobble\ekvd@stop
603         {\let\ekvd@long\long}%
604         \ekvd@extract@prefixes@prot
605         ##1\ekvd@mark\@firstofone#3\ekvd@mark\@gobble\ekvd@stop
606         {\let\ekvd@prot\protected}%
607     }%
608     \protected\def\ekvd@extract@prefixes@long##1#2##2\ekvd@mark##3##4\ekvd@stop
609     {##3}%
610     \protected\def\ekvd@extract@prefixes@prot##1#3##2\ekvd@mark##3##4\ekvd@stop
611     {##3}%
612 }

```

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.

```

613 \begingroup
614 \edef\ekvd@tmp

```

```

615   {%
616   \endgroup
617   \ekvd@extract@prefixes@
618   {\detokenize{macro:}}%
619   {\string\long}%
620   {\string\protected}%
621   }
622 \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>`.

```

623 \protected\def\ekvd@newlet#1#2%
624   {%
625   \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\let#1#2}%
626   }
627 \protected\def\ekvd@newreg#1#2%
628   {%
629   \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\csname new#2\endcsname#1}%
630   }

(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, 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.

```

631 \long\def\ekvd@assert@twoargs#1%
632   {%
633   \ekvd@ifnottwoargs{#1}{\ekvd@err@missing@definition}%
634   }
635 \long\def\ekvd@ifnottwoargs#1%
636   {%
637   \ekvd@ifempty@gtwo#1\ekv@ifempty@B
638   \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
639   }
640 \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 argument structure, respectively.

```

641 \protected\def\ekvd@assert@val#1%
642   {%
643   \ekvifdefined\ekvd@set{#1}%
644   {\expandafter\ekvd@assert@val@\csname\ekv@name\ekvd@set{#1}\endcsname}%
645   {%
646   \ekvifdefinedNoVal\ekvd@set{#1}%
647   \ekvd@err@add@val@on@noval
648   {\ekvd@err@undefined@key{#1}}%
649   \@gobble

```

```

650     }%
651   }
652 \protected\def\ekvd@assert@val@#1%
653   {%
654     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
655     \unless\ifx\ekvd@extracted@args\ekvd@one@arg@string
656       \ekvd@err@unsupported@arg
657     \fi
658     \@firstofone
659   }%
660 \protected\def\ekvd@assert@noval#1%
661   {%
662     \ekvifdefinedNoVal\ekvd@set{#1}%
663     {\expandafter\ekvd@assert@noval@\csname\ekv@name\ekvd@set{#1}N\endcsname}%
664     {%
665       \ekvifdefined\ekvd@set{#1}%
666         \ekvd@err@add@noval@on@val
667         {\ekvd@err@undefined@key{#1}}%
668       \@gobble
669     }%
670   }
671 \protected\def\ekvd@assert@noval@#1%
672   {%
673     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
674     \unless\ifx\ekvd@extracted@args\ekvd@empty
675       \ekvd@err@unsupported@arg
676     \fi
677     \@firstofone
678   }
679 \protected\def\ekvd@extract@args#1%
680   {%
681     \protected\def\ekvd@extract@args##1##2->##3\ekvd@stop
682     {\def\ekvd@extracted@args{##2}}%
683   }
684 \expandafter\ekvd@extract@args\expandafter{\detokenize{macro:}}
685 \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 686 \def\ekvd@assert@arg{\ekvd@ifnoarg\ekvd@err@missing@definition}
687 \long\def\ekvd@assert@arg@msg#1%
688   {%
689     \ekvd@ifnoarg{\ekvd@err@missing@definition@msg{#1}}%
690   }

```

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

```

\ekvd@assert@filledarg
\ekvd@ifnoarg@or@empty 691 \long\def\ekvd@assert@filledarg#1%
692   {%
693     \ekvd@ifnoarg@or@empty{#1}\ekvd@err@missing@definition
694   }

```

```

695 \long\def\ekvd@ifnoarg@or@empty#1%
696   {%
697     \ekvd@ifnoarg
698     \@firstoftwo
699     {\ekv@ifempty{#1}}%
700   }

```

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

\ekvd@assert@not@long 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.

```

\ekvd@assert@not@protected
\ekvd@assert@not@protected@also
\ekvd@assert@not@protected@also
\ekvd@assert@new
\ekvd@assert@not@new
701 \def\ekvd@assert@not@long{\ifx\ekvd@long\long\ekvd@err@no@prefix{long}\fi}
702 \def\ekvd@assert@not@protected
703   {\ifx\ekvd@prot\protected\ekvd@err@no@prefix{protected}\fi}
704 \def\ekvd@assert@not@also{\ekvd@ifalso{\ekvd@err@no@prefix{also}}{}}
705 \def\ekvd@assert@not@long@also
706   {\ifx\ekvd@long\long\ekvd@err@no@prefix@also{long}\fi}
707 \def\ekvd@assert@not@protected@also
708   {\ifx\ekvd@prot\protected\ekvd@err@no@prefix@also{protected}\fi}
709 \def\ekvd@assert@new#1#2%
710   {\csname ekvifdefined#1\endcsname\ekvd@set{#2}{\ekvd@err@not@new}}
711 \def\ekvd@assert@not@new
712   {\ifx\ekvd@ifnew\ekvd@assert@new\ekvd@err@no@prefix{new}\fi}

```

(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.

```

\ekvd@if@not@already@choice@a
\ekvd@if@not@already@choice@b
713 \protected\def\ekvd@if@not@already@choice#1%
714   {%
715     \expandafter\ekvd@if@not@already@choice@a
716     \csname\ekv@name\ekvd@set{#1}\endcsname
717     {\ekvd@h@choice\ekvd@stop}
718   }
719 \protected\def\ekvd@if@not@already@choice@a
720   {%
721     \expandafter\ekvd@if@not@already@choice@b
722   }
723 \long\protected\def\ekvd@if@not@already@choice@b#1\ekvd@h@choice#2\ekvd@stop
724   {%
725     \ekv@ifempty{#2}\@firstofone\@gobble
726   }

```

(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.

```

727 \long\def\ekvd@ifspace#1%
728   {%
729     \ekvd@ifspace@#1 \ekv@ifempty@B
730     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
731   }

```

```

732 \long\def\ekvd@ifspace@#1 % keep this space
733   {%
734     \ekv@ifempty@\ekv@ifempty@A
735   }

```

(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.

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

```

\ekvd@errm
\ekvd@err@missing@definition 736 \protected\def\ekvd@errm#1{\errmessage{expkv-def Error: #1}}
\ekvd@err@missing@definition@msg 737 \protected\def\ekvd@err@missing@definition
\ekvd@err@missing@type 738   {\ekvd@errm{Missing definition for key '\ekvd@cur'}}
\ekvd@err@undefined@prefix 739 \protected\def\ekvd@err@missing@definition@msg#1%
\ekvd@err@undefined@key 740   {\ekvd@errm{Missing definition for key '\unexpanded{#1}'}}
\ekvd@err@no@prefix 741 \protected\def\ekvd@err@missing@type
\ekvd@err@no@prefix@msg 742   {\ekvd@errm{Missing type prefix for key '\ekvd@cur'}}
\ekvd@err@no@prefix@also 743 \protected\def\ekvd@err@undefined@prefix#1%
\ekvd@err@add@val@on@noval 744   {%
\ekvd@err@add@noval@on@val 745   \ekvd@errm
\ekvd@err@unsupported@arg 746     {Undefined prefix '\unexpanded{#1}' found while processing '\ekvd@cur'}%
\ekvd@err@not@new 747   }
748 \protected\def\ekvd@err@undefined@key#1%
749   {%
750   \ekvd@errm
751     {Undefined key '\unexpanded{#1}' found while processing '\ekvd@cur'}%
752   }
753 \protected\def\ekvd@err@no@prefix#1%
754   {\ekvd@errm{prefix '#1' not accepted in '\ekvd@cur'}}
755 \protected\def\ekvd@err@no@prefix@msg#1#2%
756   {\ekvd@errm{prefix '#2' not accepted in '\unexpanded{#1}'}}
757 \protected\def\ekvd@err@no@prefix@also#1%
758   {\ekvd@errm{'\ekvd@cur' not allowed with a '#1' key}}
759 \protected\def\ekvd@err@add@val@on@noval
760   {\ekvd@errm{'\ekvd@cur' not allowed with a NoVal key}}
761 \protected\def\ekvd@err@add@noval@on@val
762   {\ekvd@errm{'\ekvd@cur' not allowed with a value taking key}}
763 \protected\def\ekvd@err@unsupported@arg\fi\@firstofone#1%
764   {%
765   \fi
766   \ekvd@errm
767     {%
768     Existing key-macro has the unsupported argument string
769     '\ekvd@extracted@args' for key '\ekvd@cur'%
770     }%
771   }
772 \protected\def\ekvd@err@not@new
773   {\ekvd@errm{The key for '\ekvd@cur' is already defined}}

```

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

`\ekvd@err@choice@invalid` The expandable error messages use `\ekvd@err`, which is just like `\ekv@err` from `expkv` or
`\ekvd@err@choice@invalid@` the way `expl3` throws expandable error messages. It uses an undefined control sequence
`\ekvd@choice@name` to start the error message. `\ekvd@err@choice@invalid` will have to use this mechanism
`\ekvd@unknown@choice@name` to throw its message. Also we have to retrieve the name parts of the choice in an easy
`\ekvd@err` way, so we use parentheses of catcode 8 here, which should suffice in most cases to allow
for a correct separation.

```

774 \def\ekvd@err@choice@invalid#1%
775   {%
776     \ekvd@err@choice@invalid@#1\ekv@stop
777   }
778 \begingroup
779 \catcode40=8
780 \catcode41=8
781 \@firstofone{\endgroup
782 \def\ekvd@choice@name#1#2#3%
783   {%
784     ekvd#1(#2)#3%
785   }
786 \def\ekvd@unknown@choice@name#1#2%
787   {%
788     ekvd:u:#1(#2)%
789   }
790 \def\ekvd@err@choice@invalid@ ekvd#1(#2)#3\ekv@stop%
791   {%
792     \ekv@ifdefined{\ekvd@unknown@choice@name{#1}{#2}}%
793       {\csname\ekvd@unknown@choice@name{#1}{#2}\endcsname{#3}}%
794       {\ekvd@err{invalid choice '#3' ('#2', set '#1')}}%
795   }
796 }
797 \begingroup
798 \edef\ekvd@err
799   {%
800     \endgroup
801     \unexpanded{\long\def\ekvd@err}##1%
802     {%
803       \unexpanded{\expandafter\ekv@err@\@firstofone}%
804       {\unexpanded\expandafter{\csname ! expkv-def Error:\endcsname}##1.}%
805       \unexpanded{\ekv@stop}%
806     }%
807   }
808 \ekvd@err

```

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

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

```
809 \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		<code>gboolpair</code>	<i>5</i>
<code>also</code>	<i>3</i>	<code>gboolpairTF</code>	<i>5</i>
<code>apptoks</code>	<i>5</i>	<code>gboolTF</code>	<i>4</i>
B		<code>gbox</code>	<i>6</i>
<code>bool</code>	<i>4</i>	<code>gdata</code>	<i>5</i>
<code>boolpair</code>	<i>5</i>	<code>gdataT</code>	<i>5</i>
<code>boolpairTF</code>	<i>5</i>	<code>gdimen</code>	<i>5</i>
<code>boolTF</code>	<i>4</i>	<code>gint</code>	<i>5</i>
<code>box</code>	<i>6</i>	<code>ginvbool</code>	<i>4</i>
C		<code>ginvboolTF</code>	<i>4</i>
<code>choice</code>	<i>6</i>	<code>gskip</code>	<i>5</i>
<code>code</code>	<i>4</i>	<code>gstore</code>	<i>5</i>
D		<code>gtoks</code>	<i>5</i>
<code>data</code>	<i>5</i>	I	
<code>dataT</code>	<i>5</i>	<code>initial</code>	<i>4</i>
<code>default</code>	<i>4</i>	<code>int</code>	<i>5</i>
<code>dimen</code>	<i>5</i>	<code>invbool</code>	<i>4</i>
E		<code>invboolTF</code>	<i>4</i>
<code>ecode</code>	<i>4</i>	L	
<code>edata</code>	<i>5</i>	<code>long</code>	<i>3</i>
<code>edataT</code>	<i>5</i>	M	
<code>edefault</code>	<i>4</i>	<code>meta</code>	<i>6</i>
<code>edimen</code>	<i>5</i>	N	
<code>einitial</code>	<i>4</i>	<code>new</code>	<i>3</i>
<code>eint</code>	<i>5</i>	<code>nmeta</code>	<i>6</i>
<code>\ekvchangeset</code>	<i>86, 92</i>	<code>noval</code>	<i>4</i>
<code>\ekvdDate</code>	<i>2, 5, 9, 15</i>	O	
<code>\ekvdef</code> <i>269, 298, 318, 337, 357, 385, 461, 572</i>		<code>oinitial</code>	<i>4</i>
<code>\ekvdefinekeys</code>	<i>2, 29</i>	P	
<code>\ekvdefNoVal</code>	<i>91, 203, 232, 580</i>	<code>protect</code>	<i>3</i>
<code>\ekvdVersion</code>	<i>2, 5, 9, 15</i>	<code>protected</code>	<i>3</i>
<code>\ekvifdefined</code>	<i>138, 160, 177, 643, 665</i>	Q	
<code>\ekvifdefinedNoVal</code>	<i>646, 662</i>	<code>qdefault</code>	<i>4</i>
<code>\ekvlet</code>	<i>128, 465</i>	S	
<code>\ekvletNoVal</code>	<i>113, 149, 168</i>	<code>set</code>	<i>6</i>
<code>\ekvparse</code>	<i>32, 469</i>	<code>skip</code>	<i>5</i>
<code>\ekvset</code>	<i>409</i>	<code>smeta</code>	<i>6</i>
<code>enoval</code>	<i>4</i>	<code>snmeta</code>	<i>6</i>
<code>eskip</code>	<i>5</i>	<code>store</code>	<i>5</i>
<code>estore</code>	<i>5</i>	T	
G		<code>TeX and L^AT_EX 2_ε commands:</code>	
<code>gapptoks</code>	<i>5</i>	<code>\ekv@err@</code>	<i>803</i>
<code>gbool</code>	<i>4</i>		

<code>\ekv@expB@unbraceA</code>	85, 90, 91, 112, 127, 148, 167, 191, 402, 433, 455	<code>\ekvd@choice@name</code>	205, 208, 234, 237, 448, 484, 503, <u>774</u>
<code>\ekv@fi@firstoftwo</code>	538	<code>\ekvd@choice@p@also</code>	521
<code>\ekv@fi@gobble</code>	625, 629	<code>\ekvd@choice@p@long</code>	<u>444</u>
<code>\ekv@gobble@mark</code>	513	<code>\ekvd@choice@p@long@</code>	<u>444</u>
<code>\ekv@ifdefined</code>	55, 58, 495, 541, 792	<code>\ekvd@choice@p@new</code>	522
<code>\ekv@ifempty</code>	80, 699, 725	<code>\ekvd@choice@p@protect</code>	<u>444</u>
<code>\ekv@ifempty@</code>	640, 734	<code>\ekvd@choice@p@protected</code>	<u>444</u>
<code>\ekv@ifempty@A</code>	638, 640, 730, 734	<code>\ekvd@choice@prefix</code>	<u>444</u>
<code>\ekv@ifempty@B</code>	637, 638, 729, 730	<code>\ekvd@choice@prefix@</code>	<u>444</u>
<code>\ekv@ifempty@false</code>	638, 730	<code>\ekvd@clear@prefixes</code>	20, 46, 477
<code>\ekv@mark</code>	49, 52, 481, 491	<code>\ekvd@cur</code> 47, 473, 478, 518, 738, 742, 746, 751, 754, 758, 760, 762, 769, 773	
<code>\ekv@name</code>	145, 165, 184, 458, 571, 579, 644, 663, 716	<code>\ekvd@empty</code>	20, 379, 674
<code>\ekv@stop</code>	49, 53, 66, 481, 493, 499, 776, 790, 805	<code>\ekvd@err</code>	<u>774</u>
<code>\ekv@strip</code>	52, 56, 491, 503	<code>\ekvd@err@add@noval@on@val</code>	666, <u>736</u>
<code>\ekvd@add@aux</code>	457, <u>567</u>	<code>\ekvd@err@add@val@on@noval</code>	647, <u>736</u>
<code>\ekvd@add@aux@</code>	<u>567</u>	<code>\ekvd@err@choice@invalid</code>	562, <u>774</u>
<code>\ekvd@add@noval</code>	85, 112, 148, 167, 423, 442, <u>567</u>	<code>\ekvd@err@choice@invalid@</code>	<u>774</u>
<code>\ekvd@add@val</code>	127, 266, 295, 316, 334, 355, 383, 419, 438, <u>567</u>	<code>\ekvd@err@missing@definition</code>	81, 633, 686, 693, <u>736</u>
<code>\ekvd@arg</code>	32, <u>34</u>	<code>\ekvd@err@missing@definition@msg</code>	473, 689, <u>736</u>
<code>\ekvd@assert@arg</code>	107, 123, 136, 158, 175, 527, 544, <u>686</u>	<code>\ekvd@err@missing@type</code>	50, 67, <u>736</u>
<code>\ekvd@assert@arg@msg</code>	478, <u>686</u>	<code>\ekvd@err@no@prefix</code>	701, 703, 704, 712, <u>736</u>
<code>\ekvd@assert@filledarg</code>	199, 260, 289, 310, 328, 349, 377, 397, <u>691</u>	<code>\ekvd@err@no@prefix@also</code> 706, 708, <u>736</u>	
<code>\ekvd@assert@new</code>	73, 537, <u>701</u>	<code>\ekvd@err@no@prefix@msg</code>	518, <u>736</u>
<code>\ekvd@assert@not@also</code>	180, 547, 704	<code>\ekvd@err@not@new</code>	541, 710, <u>736</u>
<code>\ekvd@assert@not@also@UUUUU\ekvd@assert@not@long@also</code>	701	<code>\ekvd@err@undefined@key</code>	151, 170, 187, 648, 667, <u>736</u>
<code>\ekvd@assert@not@long</code>	76, 109, 141, 163, 181, 422, 441, 446, 546, <u>701</u>	<code>\ekvd@err@undefined@prefix</code>	60, 507, <u>736</u>
<code>\ekvd@assert@not@long@also</code>	423, 442, 461	<code>\ekvd@err@unsupported@arg</code>	656, 675, <u>736</u>
<code>\ekvd@assert@not@new</code> 140, 162, 179, <u>701</u>		<code>\ekvd@errm</code>	<u>736</u>
<code>\ekvd@assert@not@protected</code>	77, 182, <u>701</u>	<code>\ekvd@extract@args</code>	<u>641</u>
<code>\ekvd@assert@not@protected@also</code>	87, <u>701</u>	<code>\ekvd@extract@prefixes</code>	585, <u>593</u>
<code>\ekvd@assert@noval</code>	577, <u>641</u>	<code>\ekvd@extract@prefixes@</code>	<u>593</u>
<code>\ekvd@assert@noval@</code>	<u>641</u>	<code>\ekvd@extract@prefixes@long</code>	<u>593</u>
<code>\ekvd@assert@twoargs</code>	243, 245, 248, 253, 429, <u>631</u>	<code>\ekvd@extract@prefixes@prot</code>	<u>593</u>
<code>\ekvd@assert@val</code>	451, 569, <u>641</u>	<code>\ekvd@extracted@args</code>	<u>641</u> , 769
<code>\ekvd@assert@val@</code>	<u>641</u>	<code>\ekvd@h@choice</code>	448, <u>553</u> , 717, 723
<code>\ekvd@choice@invalid@p</code>	511, 520, 521, 522	<code>\ekvd@h@choice@</code>	<u>553</u>
<code>\ekvd@choice@invalid@p@</code>	513, 516	<code>\ekvd@handle</code>	34
		<code>\ekvd@if@not@already@choice</code> 453, <u>713</u>	
		<code>\ekvd@if@not@already@choice@a</code>	<u>713</u>
		<code>\ekvd@if@not@already@choice@b</code>	<u>713</u>

<code>\ekvd@ifalso</code>	20 , 72 , 83 , 111 , 126 , 147 , 166 , 263 , 292 , 313 , 331 , 352 , 380 , 401 , 432 , 449 , 704	<code>\ekvd@t@box</code>	285
<code>\ekvd@ifempty<gtwo</code>	631	<code>\ekvd@t@choice</code>	444
<code>\ekvd@ifnew</code> ...	26 , 73 , 78 , 105 , 121 , 195 , 197 , 225 , 227 , 258 , 287 , 308 , 326 , 347 , 375 , 395 , 427 , 525 , 537 , 712	<code>\ekvd@t@code</code>	119
<code>\ekvd@ifnoarg</code>	36 , 41 , 99 , 686 , 697	<code>\ekvd@t@data</code>	256
<code>\ekvd@ifnoarg@or@empty</code>	691	<code>\ekvd@t@dataT</code>	256
<code>\ekvd@ifnottwoargs</code>	631	<code>\ekvd@t@default</code>	134
<code>\ekvd@ifspace</code> 48 , 65 , 480 , 498 , 511 , 514 , 727	<code>\ekvd@t@dimen</code>	345
<code>\ekvd@ifspace@</code>	727	<code>\ekvd@t@ecode</code>	119
<code>\ekvd@long</code> 20 , 69 , 125 , 269 , 298 , 318 , 337 , 357 , 385 , 417 , 461 , 572 , 603 , 701 , 706	<code>\ekvd@t@edata</code>	277
<code>\ekvd@mark</code>	602 , 605 , 608 , 610	<code>\ekvd@t@edataT</code>	282
<code>\ekvd@newlet</code>	201 , 229 , 230 , 262 , 379 , 623	<code>\ekvd@t@edefault</code>	156
<code>\ekvd@newreg</code> ...	291 , 312 , 330 , 351 , 623	<code>\ekvd@t@edimen</code>	345
<code>\ekvd@noarg</code>	32 , 34	<code>\ekvd@t@einitial</code>	173
<code>\ekvd@one@arg@string</code>	641	<code>\ekvd@t@eint</code>	345
<code>\ekvd@p@also</code>	69	<code>\ekvd@t@enoval</code>	103
<code>\ekvd@p@long</code>	69	<code>\ekvd@t@eskip</code>	345
<code>\ekvd@p@new</code>	69	<code>\ekvd@t@estore</code>	391
<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> ...	20 , 70 , 110 , 125 , 142 , 164 , 265 , 294 , 315 , 333 , 382 , 417 , 447 , 501 , 509 , 548 , 591 , 606 , 703 , 708	<code>\ekvd@t@gdimen</code>	345
<code>\ekvd@set</code>	31 , 91 , 113 , 128 , 138 , 145 , 149 , 160 , 165 , 168 , 177 , 184 , 203 , 205 , 208 , 232 , 234 , 237 , 269 , 298 , 318 , 337 , 357 , 385 , 399 , 403 , 434 , 448 , 458 , 465 , 484 , 503 , 541 , 549 , 571 , 579 , 591 , 643 , 644 , 646 , 662 , 663 , 665 , 710 , 716	<code>\ekvd@t@gint</code>	345
<code>\ekvd@set@choice</code>	484 , 503 , 530	<code>\ekvd@t@ginvbool</code>	193
<code>\ekvd@stop</code>	595 , 599 , 602 , 605 , 608 , 610 , 654 , 673 , 681 , 717 , 723	<code>\ekvd@t@ginvboolTF</code>	193
<code>\ekvd@t@apptoks</code>	324	<code>\ekvd@t@meta</code>	393
<code>\ekvd@t@bool</code>	193	<code>\ekvd@t@nmeta</code>	393
<code>\ekvd@t@boolpair</code>	223	<code>\ekvd@t@noval</code>	103
<code>\ekvd@t@boolpairTF</code>	223	<code>\ekvd@t@oinitial</code>	173
<code>\ekvd@t@boolTF</code>	193	<code>\ekvd@t@qdefault</code>	134
		<code>\ekvd@t@set</code>	74
		<code>\ekvd@t@skip</code>	345
		<code>\ekvd@t@smeta</code>	425
		<code>\ekvd@t@snmeta</code>	425
		<code>\ekvd@t@store</code>	373

