

# exp<sub>k</sub>v<sub>DEF</sub>

a key-defining frontend for exp<sub>k</sub>v

Jonathan P. Spratte\*

2022-01-29 v0.9

## Abstract

exp<sub>k</sub>v<sub>DEF</sub> provides a small  $\langle key \rangle = \langle value \rangle$  interface to define keys for exp<sub>k</sub>v. Key-types are declared using prefixes, similar to static typed languages. The stylised name is exp<sub>k</sub>v<sub>DEF</sub> but the files use exp<sub>k</sub>v-def, this is due to CTAN-rules which don't allow | in package names since that is the pipe symbol in \*nix shells.

## Contents

<b>1</b>	<b>Documentation</b>	<b>2</b>
1.1	Macros	2
1.2	Prefixes	2
1.2.1	p-Prefixes	2
1.2.2	t-Prefixes	3
1.3	Bugs	8
1.4	Example	9
1.5	License	10
<b>2</b>	<b>Implementation</b>	<b>11</b>
2.1	The L <sup>A</sup> T <sub>E</sub> X Package	11
2.2	The ConT <sub>E</sub> Xt module	11
2.3	The Generic Code	11
2.3.1	Key Types	14
2.3.2	Key Type Helpers	28
2.3.3	Handling also	29
2.3.4	Tests	30
2.3.5	Messages	33
	<b>Index</b>	<b>36</b>

---

\*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, as a  $\LaTeX$  package, and as a `ConTeXt` module. It'll automatically load `expkv` in the same mode as well. To use it, just use one of

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

### 1.1 Macros

Apart from version and date containers there is only a single user-facing macro, and that should be used to define keys.

---

```
\ekvdefinekeys \ekvdefinekeys{\set}\{\langle key \rangle = \langle value \rangle, ...}
```

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
	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 $\backslashedef$ .	
noval enoval	noval $\langle key \rangle = \{ \langle definition \rangle \}$	new also protected long
	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 $\backslashedef$ .	
default qdefault odefault fdefault edefault	default $\langle key \rangle = \{ \langle definition \rangle \}$	new also protected long
	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. <sup>1</sup> The edefault on the other hand fully expands the $\langle key \rangle$ -code with $\langle definition \rangle$ as its argument inside of an $\backslashedef$ .	
initial oinitial finitial einitial	initial $\langle key \rangle = \{ \langle value \rangle \}$	new also protected long
	initial $\langle key \rangle$	
	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. <sup>2</sup>	
	If you don't provide a value (and no equals sign) a noval $\langle key \rangle$ of the same name is called once (or, if you specified a default for a value taking key that would be used).	

<sup>1</sup>For those familiar with T<sub>E</sub>X-coding: This uses a  $\backslashromannumeral$ -expansion.

<sup>2</sup>Again using  $\backslashromannumeral$ .

<b>bool</b> <b>gbool</b> <b>boolTF</b> <b>gboolTF</b>	<b>bool</b> $\langle key \rangle = \langle cs \rangle$ <span style="float: right;">new also protected <b>long</b></span> The $\langle cs \rangle$ should be a single control sequence, such as <code>\iffoo</code> . This will define $\langle key \rangle$ to be a boolean key, which only takes the values <code>true</code> or <code>false</code> and will throw an error for other values. If the key is used without a $\langle value \rangle$ it'll have the same effect as if you use $\langle key \rangle = true$ . <code>bool</code> and <code>gbool</code> will behave like T <sub>E</sub> X-ifs so either be <code>\iftrue</code> or <code>\iffalse</code> . The <code>boolTF</code> and <code>gboolTF</code> variants will both take two arguments and if true the first will be used else the second, so they are always either <code>\@firstoftwo</code> or <code>\@secondoftwo</code> . The variants with a leading <code>g</code> will set the control sequence globally, the others locally. If $\langle cs \rangle$ is not yet defined it'll be initialised as the <code>false</code> version. Note that the initialisation is <i>not</i> done with <code>\newif</code> , so you will not be able to do <code>\footrue</code> outside of the $\langle key \rangle = \langle value \rangle$ interface, but you could use <code>\newif</code> yourself. Even if the $\langle key \rangle$ will not be <code>\protected</code> the commands which execute the <code>true</code> or <code>false</code> choice will be, so the usage should be safe in an expansion context (e.g., you can use <code>edefault \langle key \rangle = false</code> without an issue to change the default behaviour to execute the <code>false</code> choice). Internally a <code>bool \langle key \rangle</code> is the same as a choice key which is set up to handle <code>true</code> and <code>false</code> as choices.
<b>invbool</b> <b>ginvbool</b> <b>invboolTF</b> <b>ginvboolTF</b>	<b>bool</b> $\langle key \rangle = \langle cs \rangle$ <span style="float: right;">new also protected <b>long</b></span> 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.
<b>boolpair</b> <b>gboolpair</b> <b>boolpairTF</b> <b>gboolpairTF</b>	<b>boolpair</b> $\langle key \rangle = \langle cs_1 \rangle \langle cs_2 \rangle$ <span style="float: right;">new also protected <b>long</b></span> 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.
<b>store</b> <b>estore</b> <b>gstore</b> <b>xstore</b>	<b>store</b> $\langle key \rangle = \langle cs \rangle$ <span style="float: right;">new also protected <b>long</b></span> 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> .
<b>data</b> <b>edata</b> <b>gdata</b> <b>xdata</b>	<b>data</b> $\langle key \rangle = \langle cs \rangle$ <span style="float: right;">new also protected <b>long</b></span> 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> .
<b>dataT</b> <b>edataT</b> <b>gdataT</b> <b>xdataT</b>	<b>dataT</b> $\langle key \rangle = \langle cs \rangle$ <span style="float: right;">new also protected <b>long</b></span> 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 <code>{\langle value \rangle}</code> .

int eint gint xint	int $\langle key \rangle = \langle cs \rangle$	new also protected long
dimen edimen gdimen xdimen	dimen $\langle key \rangle = \langle cs \rangle$	new also protected long
skip eskip gskip xskip	skip $\langle key \rangle = \langle cs \rangle$	new also protected long
toks gtoks apptoks gapptoks pretoks gpretoks	toks $\langle key \rangle = \langle cs \rangle$	new also protected long
box gbox	box $\langle key \rangle = \langle cs \rangle$	new also protected long
meta	meta $\langle key \rangle = \{\langle key \rangle = \langle value \rangle, \dots\}$	new also protected long
nmeta	nmeta $\langle key \rangle = \{\langle key \rangle = \langle value \rangle, \dots\}$	new also protected long
smeta	smeta $\langle key \rangle = \{\langle set \rangle\}\{\langle key \rangle = \langle value \rangle, \dots\}$	new also protected long
snmeta	snmeta $\langle key \rangle = \{\langle set \rangle\}\{\langle key \rangle = \langle value \rangle, \dots\}$	new also protected long

---

**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.

---

**choice-store** `choice-store <key> = <cs>{<value>, ...}` new also **protected long**

This defines a special type of choice key that'll store the given choice inside the macro `<cs>` (so `<cs>` should be a single control sequence name such as `\foo`). Since storing inside a macro can't be done expandably every choice-code is `\protected`, you might define the `choice-store` key itself as `protected` as well if you want. Since the definition of each choice is predefined with this key type the choice list should just be a comma separated list of valid choices.

This means that the following choice and choice-store keys are equivalent at use time:

```
\newcommand*\mya{ }
\ekvdefinekeys{example}
{
  choice key1 = {a=\def\mya{a}, b=\def\mya{b}, c=\def\mya{c} }
  ,choice-store key2 = \mya{a,b,c}
}
```

---

**choice-enum** `choice-enum <key> = <cs>{<value>, ...}` new also **protected long**

This is similar to `choice-store`, the differences are: `<cs>` should be a count-register or is initialised as such if the `<cs>` is undefined (via `\newcount`); instead of the value the position of the value in the given list is stored in this register (zero-based).

This means that the following choice and choice-enum keys are equivalent at use time:

```
\newcount\myb
\ekvdefinekeys{example}
{
  choice key1 = {a={\myb=0 }, b={\myb=1 }, c={\myb=2 }}
  ,choice-enum key2 = \myb{a,b,c}
}
```

---

`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.<sup>3</sup>

---

`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](https://github.com/Skillmon/tex_expkv-def)

---

<sup>3</sup>There 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 © 2020–2022 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<sup>A</sup>T<sub>E</sub>X Package

Just like for `expkv` we provide a small L<sup>A</sup>T<sub>E</sub>X package that sets up things such that we behave nicely on L<sup>A</sup>T<sub>E</sub>X 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 ConT<sub>E</sub>Xt module

```
10 \writestatus{loading}{ConTEXt User Module / expkv-def}
11 \usemodule{expkv}
12 \unprotect
13 \input expkv-def.tex
14 \writestatus{loading}
15   {ConTEXt User Module / expkv-def / Version \ekvdVersion\space loaded}
16 \protect\endinput
```

### 2.3 The Generic Code

The rest of this implementation will be the generic code.

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

```
17 \input expkv
18   We make sure that expkv-def.tex is only input once:
19 \expandafter\ifx\csname ekvdVersion\endcsname\relax
20 \else
21   \expandafter\endinput
22 \fi
```

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

```
\ekvdDate
22 \def\ekvdVersion{0.9}
23 \def\ekvdDate{2022-01-29}
```

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

If the L<sup>A</sup>T<sub>E</sub>X 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.

```
24 \csname ekvd@tmp\endcsname
25   Store the category code of @ to later be able to reset it and change it to 11 for now.
26 \expandafter\chardef\csname ekvd@tmp\endcsname=\catcode'\@
27 \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 `expkVDEF` was initialized.

`\ekvd@ifprimitive`

```

27 \protected\long\def\ekvd@ifprimitive#1%
28   {%
29     \begingroup
30     \edef\ekvd@tmpa{\string #1}%
31     \edef\ekvd@tmpb{\meaning#1}%
32     \expandafter
33     \endgroup
34     \ifx\ekvd@tmpa\ekvd@tmpb
35     \ekv@fi@firstoftwo
36     \fi
37     \@secondoftwo
38   }

```

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

`\ekvd@long` `expkVDEF` 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, `\ekvd@clear@prefixes` `ifalso` will be `\@secondoftwo`, and `ifnew` will just use its third argument.

```

39 \protected\def\ekvd@clear@prefixes
40   {%
41     \let\ekvd@long\ekv@empty
42     \let\ekvd@prot\ekv@empty
43     \let\ekvd@ifalso\@secondoftwo
44     \long\def\ekvd@ifnew##1##2##3{##3}%
45   }
46 \ekvd@clear@prefixes

```

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

`\ekvdefinekeys` This is the one front-facing macro which provides the interface to define keys. It's using `\ekvparse` to handle the `<key>=<value>` list, the interpretation will be done by `\ekvd@noarg` and `\ekvd@`. The `<set>` for which the keys should be defined is stored in `\ekvd@set`.

```

47 \protected\def\ekvdefinekeys#1%
48   {%
49     \def\ekvd@set{#1}%
50     \ekvparse\ekvd@noarg\ekvd@arg
51   }

```

*(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 `<key>=<value>` 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.

```

52 \protected\long\def\ekvd@noarg#1%
53   {%
54     \let\ekvd@ifnoarg\@firstoftwo
55     \expandafter\ekvd@handle\detokenize{#1}\ekvd@stop{}%
56   }

```

```

57 \protected\long\def\ekvd@arg#1%
58   {%
59     \let\ekvd@ifnoarg\@secondoftwo
60     \expandafter\ekvd@handle\detokenize{#1}\ekvd@stop
61   }
62 \protected\long\def\ekvd@handle#1\ekvd@stop#2%
63   {%
64     \ekvd@clear@prefixes
65     \def\ekvd@cur{#1}%
66     \ekvd@ifspace{#1}%
67     {\ekvd@prefix\ekv@mark#1\ekv@stop{#2}}%
68     \ekvd@err@missing@type
69   }

```

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

`\ekvd@prefix` **EXPANDDEF** separates prefixes into two groups, the first being prefixes in the T<sub>E</sub>X sense (`\long` and `\protected`) which use `@p@` in their name, the other being key-types (`code`, `int`, *etc.*) which use `@t@` instead. `\ekvd@prefix` splits at the first space and checks whether its a `@p@` or `@t@` type prefix. If it is neither throw an error and gobble the definition (the value).

```

70 \protected\def\ekvd@prefix#1 {\ekv@strip{#1}\ekvd@prefix@\ekv@mark}
71 \protected\def\ekvd@prefix@#1#2\ekv@stop
72   {%
73     \ekv@ifdefined{ekvd@t@#1}%
74     {\ekv@strip{#2}{\csname ekvd@t@#1\endcsname}}%
75     {%
76       \ekv@ifdefined{ekvd@p@#1}%
77       {\csname ekvd@p@#1\endcsname\ekvd@prefix@after@p{#2}}%
78       {\ekvd@err@undefined@prefix{#1}\@gobble}%
79     }%
80   }

```

(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.

```

81 \protected\def\ekvd@prefix@after@p#1%
82   {%
83     \ekvd@ifspace{#1}%
84     {\ekvd@prefix#1\ekv@stop}%
85     {\ekvd@err@missing@type\@gobble}%
86   }

```

(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`

```

87 \protected\def\ekvd@p@long{\let\ekvd@long\long}
88 \protected\def\ekvd@p@protected{\let\ekvd@prot\protected}
89 \let\ekvd@p@protect\ekvd@p@protected
90 \protected\def\ekvd@p@also{\let\ekvd@ifalso\@firstoftwo}
91 \protected\def\ekvd@p@new{\let\ekvd@ifnew\ekvd@assert@new}

```

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

### 2.3.1 Key Types

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

`\ekvd@t@set`

```

92 \protected\def\ekvd@type@set#1#2%
93   {%
94     \ekvd@assert@not@long
95     \ekvd@assert@not@protected
96     \ekvd@ifnew{NoVal}{#1}%
97     {%
98       \ekv@ifempty{#2}%
99       {\ekvd@err@missing@definition}%
100      {%
101        \ekvd@ifalso
102        {%
103          \ekv@expargtwice{\ekvd@add@noval{#1}}%
104          {\ekvchangeset{#2}}%
105          \ekvd@assert@not@protected@also
106        }%
107        {\ekv@expargtwice{\ekvdefNoVal\ekvd@set{#1}}{\ekvchangeset{#2}}}%
108      }%
109    }%
110  }
111 \protected\def\ekvd@t@set#1#2%
112   {%
113     \ekvd@ifnoarg
114     {\ekvd@type@set{#1}{#1}}%
115     {\ekvd@type@set{#1}{#2}}%
116  }

```

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

`\ekvd@type@noval`

`\ekvd@t@noval`

`\ekvd@t@enoval`

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

```

117 \protected\long\def\ekvd@type@noval#1#2#3%
118   {%
119     \ekvd@ifnew{NoVal}{#2}%
120     {%
121       \ekvd@assert@arg
122       {%
123         \ekvd@assert@not@long
124         \ekvd@prot#1\ekvd@tmp{#3}%
125         \ekvd@ifalso
126         {\ekv@exparg{\ekvd@add@noval{#2}}\ekvd@tmp{}}%
127         {\ekvletNoVal\ekvd@set{#2}\ekvd@tmp}%
128       }%
129     }%
130  }
131 \protected\def\ekvd@t@noval{\ekvd@type@noval\def}
132 \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`.

```

\ekvd@t@code
\ekvd@t@ecode
133 \protected\long\def\ekvd@type@code#1#2#3%
134   {%
135     \ekvd@ifnew{#2}%
136     {%
137       \ekvd@assert@arg
138       {%
139         \ekvd@prot\ekvd@long#1\ekvd@tmp##1{#3}%
140         \ekvd@ifalso
141         {\ekv@exparg{\ekvd@add@val{#2}}{\ekvd@tmp{##1}}{}}%
142         {\ekvlet\ekvd@set{#2}\ekvd@tmp}%
143       }%
144     }%
145   }
146 \protected\def\ekvd@t@code{\ekvd@type@code\def}
147 \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.

```

148 \protected\long\def\ekvd@type@default#1#2#3#4%
149   {%
150     \ekvd@assert@arg
151     {%
152       \ekvifdefined\ekvd@set{#3}%
153       {%
154         \ekvd@assert@not@new
155         \ekvd@assert@not@long
156         \ekvd@prot\edef\ekvd@tmp
157         {%
158           \ekv@unexpanded\expandafter#1%
159           {#2\csname\ekv@name\ekvd@set{#3}\endcsname{#4}}%
160         }%
161       \ekvd@ifalso
162       {\ekv@exparg{\ekvd@add@noval{#3}}\ekvd@tmp{}}%
163       {\ekvletNoVal\ekvd@set{#3}\ekvd@tmp}%
164     }%
165     {\ekvd@err@undefined@key{#3}}%
166   }%
167 }
168 \protected\def\ekvd@t@default{\ekvd@type@default{}}
169 \protected\def\ekvd@t@qdefault{\ekvd@type@default{\expandafter\expandafter}}
170 \let\ekvd@t@odefault\ekvd@t@qdefault
171 \protected\def\ekvd@t@fdefault{\ekvd@type@default{\romannumeral'\^^@}}

```

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

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

```

172 \protected\long\def\ekvd@t@edefault#1#2%
173   {%
174     \ekvd@assert@arg
175     {%
176       \ekvifdefined\ekvd@set{#1}%
177       {%
178         \ekvd@assert@not@new
179         \ekvd@assert@not@long
180         \ekvd@prot\edef\ekvd@tmp
181           {\csname\ekv@name\ekvd@set{#1}\endcsname{#2}}%
182         \ekvd@ifalso
183           {\ekv@exparg{\ekvd@add@noval{#1}\ekvd@tmp}}}%
184           {\ekvletNoVal\ekvd@set{#1}\ekvd@tmp}}%
185       }%
186     {\ekvd@err@undefined@key{#1}}%
187   }%
188 }

```

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

```

\ekvd@t@initial
\ekvd@t@oinitial
\ekvd@t@finitial
\ekvd@t@einitial
189 \long\def\ekvd@type@initial#1#2#3#4%
190   {%
191     \ekvd@assert@not@new
192     \ekvd@assert@not@also
193     \ekvd@assert@not@long
194     \ekvd@assert@not@protected
195     \ekvd@ifnoarg
196     {%
197       \ekvifdefinedNoVal\ekvd@set{#3}%
198       {\csname\ekv@name\ekvd@set{#3}N\endcsname}%
199       {\ekvd@err@undefined@noval{#3}}%
200     }%
201     {%
202       \ekvifdefined\ekvd@set{#3}%
203       {%
204         #1{#2#4}%
205         \csname\ekv@name\ekvd@set{#3}\expandafter\endcsname\expandafter
206           {\ekvd@tmp}}%
207       }%
208     {\ekvd@err@undefined@key{#3}}%
209   }%
210 }
211 \def\ekvd@t@initial{\ekvd@type@initial{\def\ekvd@tmp}{}}
212 \def\ekvd@t@oinitial{\ekvd@type@initial{\ekv@exparg{\def\ekvd@tmp}{}}}
213 \def\ekvd@t@einitial{\ekvd@type@initial{\edef\ekvd@tmp}{}}
214 \def\ekvd@t@finitial
215   {\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
\ekvd@t@gboolTF
\ekvd@t@invbool
\ekvd@t@ginvbool
\ekvd@t@invboolTF
\ekvd@t@ginvboolTF

```



```

216 \protected\def\ekvd@type@bool#1#2#3#4#5%
217   {%
218     \ekvd@ifnew{}{#4}%
219     {%
220       \ekvd@ifnew{NoVal}{#4}%
221       {%
222         \ekvd@assert@filledarg{#5}%
223         {%
224           \ekvd@newlet#5#3%
225           \ekvd@type@choice{#4}%
226           \protected\ekvdefNoVal\ekvd@set{#4}{#1\let#5#2}%
227           \protected\expandafter\def
228             \csname\ekvd@choice@name\ekvd@set{#4}{true}\endcsname
229             {#1\let#5#2}%
230           \protected\expandafter\def
231             \csname\ekvd@choice@name\ekvd@set{#4}{false}\endcsname
232             {#1\let#5#3}%
233         }%
234       }%
235     }%
236   }
237 \protected\def\ekvd@t@bool{\ekvd@type@bool}\iftrue\iffalse}
238 \protected\def\ekvd@t@gbool{\ekvd@type@bool\global\iftrue\iffalse}
239 \protected\def\ekvd@t@boolTF{\ekvd@type@bool}\@firstoftwo\@secondoftwo}
240 \protected\def\ekvd@t@gboolTF{\ekvd@type@bool\global\@firstoftwo\@secondoftwo}
241 \protected\def\ekvd@t@invbool{\ekvd@type@bool}\iffalse\iftrue}
242 \protected\def\ekvd@t@invboolTF{\ekvd@type@bool\global\iffalse\iftrue}
243 \protected\def\ekvd@t@invboolTF{\ekvd@type@bool}\@secondoftwo\@firstoftwo}
244 \protected\def\ekvd@t@ginvboolTF
245   {\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
246 \protected\def\ekvd@type@boolpair#1#2#3#4#5#6%
247   {%
248     \ekvd@ifnew{}{#4}%
249     {%
250       \ekvd@ifnew{NoVal}{#4}%
251       {%
252         \ekvd@newlet#5#3%
253         \ekvd@newlet#6#2%
254         \ekvd@type@choice{#4}%
255         \protected\ekvdefNoVal\ekvd@set{#4}{#1\let#5#2#1\let#6#3}%
256         \protected\expandafter\def
257           \csname\ekvd@choice@name\ekvd@set{#4}{true}\endcsname
258           {#1\let#5#2#1\let#6#3}%
259         \protected\expandafter\def
260           \csname\ekvd@choice@name\ekvd@set{#4}{false}\endcsname
261           {#1\let#5#3#1\let#6#2}%
262       }%
263     }%
264   }

```

```

265 \protected\def\ekvd@t@boolpair#1#2%
266   {\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair{}}\iftrue\iffalse{#1}#2}}
267 \protected\def\ekvd@t@gboolpair#1#2%
268   {\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair\global\iftrue\iffalse{#1}#2}}
269 \protected\def\ekvd@t@boolpairTF#1#2%
270   {%
271     \ekvd@assert@twoargs{#2}%
272     {\ekvd@type@boolpair{}}\@firstoftwo\@secondoftwo{#1}#2}%
273   }
274 \protected\def\ekvd@t@gboolpairTF#1#2%
275   {%
276     \ekvd@assert@twoargs{#2}%
277     {\ekvd@type@boolpair\global\@firstoftwo\@secondoftwo{#1}#2}%
278   }

```

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

```

\ekvd@type@data
  \ekvd@t@data
  \ekvd@t@gdata
  \ekvd@t@dataT
  \ekvd@t@gdataT
279 \protected\def\ekvd@type@data#1#2#3#4#5#6%
280   {%
281     \ekvd@ifnew{#5}%
282     {%
283       \ekvd@assert@filledarg{#6}%
284       {%
285         \ekvd@newlet#6#1%
286         \ekvd@ifalso
287         {%
288           \let\ekvd@prot\protected
289           \ekvd@add@val{#5}{\long#2#6###1#3{###1{#4}}}{#5}%
290         }%
291         {%
292           \protected\ekvd@long\ekvddef\ekvd@set{#5}%
293             {\long#2#6###1#3{###1{#4}}}%
294         }%
295       }%
296     }%
297   }
298 \protected\def\ekvd@t@data
299   {\ekvd@type@data\@secondoftwo\edef{###2}{\ekv@unexpanded{##1}}}
300 \protected\def\ekvd@t@edata{\ekvd@type@data\@secondoftwo\edef{###2}{##1}}
301 \protected\def\ekvd@t@gdata
302   {\ekvd@type@data\@secondoftwo\xdef{###2}{\ekv@unexpanded{##1}}}
303 \protected\def\ekvd@t@xdata{\ekvd@type@data\@secondoftwo\xdef{###2}{##1}}
304 \protected\def\ekvd@t@dataT
305   {\ekvd@type@data\@gobble\edef{#}{\ekv@unexpanded{##1}}}
306 \protected\def\ekvd@t@edataT{\ekvd@type@data\@gobble\edef{#}{##1}}
307 \protected\def\ekvd@t@gdataT
308   {\ekvd@type@data\@gobble\xdef{#}{\ekv@unexpanded{##1}}}
309 \protected\def\ekvd@t@xdataT{\ekvd@type@data\@gobble\xdef{#}{##1}}

```

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

```

\ekvd@type@box
  \ekvd@t@box
  \ekvd@t@gbox

```

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

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

```

310 \protected\def\ekvd@type@box#1#2#3%
311   {%
312     \ekvd@ifnew{#2}%
313     {%
314       \ekvd@assert@filledarg{#3}%
315       {%
316         \ekvd@newreg#3{box}%
317         \ekvd@ifalso
318         {%
319           \let\ekvd@prot\protected
320           \ekvd@add@val{#2}{#1\setbox#3=\hbox{\begingroup##1\endgroup}}{ }%
321         }%
322         {%
323           \protected\ekvd@long\ekvdef\ekvd@set{#2}%
324             {#1\setbox#3=\hbox{\begingroup##1\endgroup}}%
325         }%
326       }%
327     }%
328   }
329 \protected\def\ekvd@t@box{\ekvd@type@box{}}
330 \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
331 \protected\def\ekvd@type@toks#1#2#3%
332   {%
333     \ekvd@ifnew{#2}%
334     {%
335       \ekvd@assert@filledarg{#3}%
336       {%
337         \ekvd@newreg#3{toks}%
338         \ekvd@ifalso
339         {%
340           \let\ekvd@prot\protected
341           \ekvd@add@val{#2}{#1#3={##1}}{ }%
342         }%
343         {\protected\ekvd@long\ekvdef\ekvd@set{#2}{#1#3={##1}}}%
344       }%
345     }%
346   }
347 \protected\def\ekvd@t@toks{\ekvd@type@toks{}}
348 \protected\def\ekvd@t@gtoks{\ekvd@type@toks\global}

```

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

`\ekvd@type@preapptoks` Just like `toks`, but expand the current contents of the `toks` register to append the new contents.

```

\ekvd@t@apptoks
\ekvd@t@gapptoks
349 \ekvd@ifprimitive\toksapp
350   {%
351     \protected\def\ekvd@type@preapptoks#1#2#3%
352     {%

```

```

353     \ekvd@ifnew{#2}%
354     {%
355         \ekvd@assert@filledarg{#3}%
356         {%
357             \ekvd@newreg#3{toks}%
358             \ekvd@ifalso
359             {%
360                 \let\ekvd@prot\protected
361                 \ekvd@add@val{#2}{#1#3{##1}}{}%
362             }%
363             {\protected\ekvd@long\ekvdef\ekvd@set{#2}{#1#3{##1}}}%
364         }%
365     }%
366 }
367 \protected\def\ekvd@t@apptoks{\ekvd@type@preapptoks\toksapp}
368 \protected\def\ekvd@t@gapptoks{\ekvd@type@preapptoks\gtoksapp}
369 \protected\def\ekvd@t@pretoks{\ekvd@type@preapptoks\tokspre}
370 \protected\def\ekvd@t@gpretoks{\ekvd@type@preapptoks\gtokspre}
371 }
372 {%
373 \protected\def\ekvd@type@apptoks#1#2#3%
374     {%
375         \ekvd@ifnew{#2}%
376         {%
377             \ekvd@assert@filledarg{#3}%
378             {%
379                 \ekvd@newreg#3{toks}%
380                 \ekvd@ifalso
381                 {%
382                     \let\ekvd@prot\protected
383                     \ekvd@add@val{#2}{#1#3=\expandafter{\the#3##1}}{}%
384                 }%
385                 {%
386                     \protected\ekvd@long\ekvdef\ekvd@set{#2}%
387                     {#1#3=\expandafter{\the#3##1}}%
388                 }%
389             }%
390         }%
391     }
392 \protected\def\ekvd@t@apptoks{\ekvd@type@apptoks{}}
393 \protected\def\ekvd@t@gapptoks{\ekvd@type@apptoks\global}
394 \newtoks\ekvd@toks
395 \protected\def\ekvd@type@pretoks#1#2#3%
396     {%
397         \ekvd@ifnew{#2}%
398         {%
399             \ekvd@assert@filledarg{#3}%
400             {%
401                 \ekvd@newreg#3{toks}%
402                 \ekvd@ifalso
403                 {%
404                     \let\ekvd@prot\protected
405                     \ekvd@add@val{#2}%
406                     {%

```

```

407         \ekvd@toks={##1}%
408         #1#3=\expandafter{\the\expandafter\ekvd@toks\the#3}%
409     }%
410     {%
411 }%
412 {%
413     \protected\ekvd@long\ekvdef\ekvd@set{#2}%
414     {%
415         \ekvd@toks={##1}%
416         #1#3=\expandafter{\the\expandafter\ekvd@toks\the#3}%
417     }%
418 }%
419 }%
420 }%
421 }
422 \protected\def\ekvd@t@pretoks{\ekvd@type@pretoks{}}
423 \protected\def\ekvd@t@gpretoks{\ekvd@type@pretoks\global}
424 }

```

(End definition for `\ekvd@type@preapptoks`, `\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>`.

```

425 \protected\def\ekvd@type@reg#1#2#3#4#5#6%
426     {%
427     \ekvd@ifnew{#5}%
428     {%
429     \ekvd@assert@filledarg{#6}%
430     {%
431     \ekvd@newreg#6{#1}%
432     \ekvd@ifalso
433     {%
434     \let\ekvd@prot\protected
435     \ekvd@add@val{#5}{#2#6=#3##1#4\relax}{}}%
436     }%
437     {\protected\ekvd@long\ekvdef\ekvd@set{#5}{#2#6=#3##1#4\relax}}%
438     }%
439 }%
440 }
441 \protected\def\ekvd@t@int{\ekvd@type@reg{count}{-}{-}}
442 \protected\def\ekvd@t@eint{\ekvd@type@reg{count}{-}\numexpr\relax}
443 \protected\def\ekvd@t@gint{\ekvd@type@reg{count}\global{}}
444 \protected\def\ekvd@t@xint{\ekvd@type@reg{count}\global\numexpr\relax}
445 \protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{-}{-}}
446 \protected\def\ekvd@t@edimen{\ekvd@type@reg{dimen}{-}\dimexpr\relax}
447 \protected\def\ekvd@t@gdimen{\ekvd@type@reg{dimen}\global{}}
448 \protected\def\ekvd@t@xdimen{\ekvd@type@reg{dimen}\global\dimexpr\relax}
449 \protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{-}{-}}
450 \protected\def\ekvd@t@eskip{\ekvd@type@reg{skip}{-}\glueexpr\relax}
451 \protected\def\ekvd@t@gskip{\ekvd@type@reg{skip}\global{}}
452 \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  
`\ekvd@t@store` also store # easily.  
`\ekvd@t@gstore`

```

453 \protected\def\ekvd@type@store#1#2#3#4%
454   {%
455     \ekvd@ifnew{#3}%
456     {%
457       \ekvd@assert@filledarg{#4}%
458       {%
459         \ekvd@newlet#4\ekv@empty
460         \ekvd@ifalso
461         {%
462           \let\ekvd@prot\protected
463           \ekvd@add@val{#3}{#1#4{#2}}}%
464         }%
465       }\protected\ekvd@long\ekvdef\ekvd@set{#3}{#1#4{#2}}}%
466     }%
467   }%
468 }
469 \protected\def\ekvd@t@store{\ekvd@type@store\edef{\ekv@unexpanded{##1}}}
470 \protected\def\ekvd@t@gstore{\ekvd@type@store\xdef{\ekv@unexpanded{##1}}}
471 \protected\def\ekvd@t@estore{\ekvd@type@store\edef{##1}}
472 \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,  
`\ekvd@type@meta@a` with the same `<set>`.

```

473 \protected\long\def\ekvd@type@meta#1#2#3#4#5#6#7%
474   {%
475     \ekvd@ifnew{#1}{#6}%
476     {%
477       \ekvd@assert@filledarg{#7}%
478       {%
479         \edef\ekvd@tmp{\ekvd@set}%
480         \expandafter\ekvd@type@meta@a\expandafter{\ekvd@tmp}{#7}{#2}%
481         \ekvd@ifalso
482         {\ekv@exparg{#3{#6}}{\ekvd@tmp#4}{#5}}%
483         {\csname ekvlet#1\endcsname\ekvd@set{#6}\ekvd@tmp}%
484       }%
485     }%
486   }
487 \protected\long\def\ekvd@type@meta@a#1#2%
488   {%
489     \expandafter\ekvd@type@meta@b\expandafter{\ekvset{#1}{#2}}%
490   }
491 \protected\def\ekvd@type@meta@b
492   {%
493     \expandafter\ekvd@type@meta@c\expandafter
494   }
495 \protected\long\def\ekvd@type@meta@c#1#2%
496   {%
497     \ekvd@prot\ekvd@long\def\ekvd@tmp#2{#1}%
498   }
499 \protected\def\ekvd@t@meta{\ekvd@type@meta}{##1}\ekvd@add@val{{##1}}{}}

```

```

500 \protected\def\ekvd@t@meta
501   {%
502     \ekvd@assert@not@long
503     \ekvd@type@meta{NoVal}-{\ekvd@add@noval}\ekvd@assert@not@long@also
504   }

```

(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
505 \protected\long\def\ekvd@type@smeta#1#2#3#4#5#6#7%
506   {%
507     \ekvd@ifnew{#1}{#6}%
508     {%
509       \ekvd@assert@twoargs{#7}%
510       {%
511         \ekvd@type@meta@a#7{#2}%
512         \ekvd@ifalso
513         {\ekv@exparg{#3{#6}}{\ekvd@tmp#4}{#5}}%
514         {\csname ekvlet#1\endcsname\ekvd@set{#6}\ekvd@tmp}%
515       }%
516     }%
517   }
518 \protected\def\ekvd@t@smeta{\ekvd@type@smeta}{##1}\ekvd@add@val{##1}{-}
519 \protected\def\ekvd@t@snmeta
520   {%
521     \ekvd@assert@not@long
522     \ekvd@type@smeta{NoVal}-{\ekvd@add@noval}\ekvd@assert@not@long@also
523   }

```

(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.

```

\ekvd@type@choice
\ekvd@populate@choice
\ekvd@populate@choice@
\ekvd@populate@choice@noarg
\ekvd@choice@prefix
\ekvd@choice@prefix@
\ekvd@choice@prefix@done
\ekvd@choice@p@protected
\ekvd@choice@p@protect
\ekvd@choice@p@long
\ekvd@choice@p@long@
\ekvd@t@choice
524 \protected\def\ekvd@type@choice#1%
525   {%
526     \ekvd@assert@not@long
527     \ekv@expargtwice{\ekvd@prot\def\ekvd@tmp##1}%
528     {%
529       \expandafter\expandafter\expandafter
530       \ekvd@h@choice
531       \expandafter\expandafter\expandafter
532       {\expandafter\ekvd@choice@name\expandafter{\ekvd@set}{#1}{##1}}%
533     }%
534     \ekvd@ifalso
535     {%
536       \ekvd@assert@val{#1}%
537       {%
538         \ekvd@if@not@already@choice{#1}%
539         {%
540           \ekv@exparg

```

```

541         {%
542             \expandafter\ekvd@add@aux
543             \csname\ekv@name\ekvd@set{#1}\endcsname{##1}{#1}%
544         }%
545         {\ekvd@tmp{##1}}%
546         {\ekvd@long\ekvdef}\ekvd@assert@not@long@also
547     }%
548 }%
549 }%
550 {\ekvlet\ekvd@set{#1}\ekvd@tmp}%
551 }

```

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

```

552 \protected\def\ekvd@populate@choice
553     {%
554     \ekvparse\ekvd@populate@choice@noarg\ekvd@populate@choice@
555     }
556 \protected\long\def\ekvd@populate@choice@noarg#1%
557     {%
558     \expandafter\ekvd@err@missing@definition@msg\expandafter{\ekvd@cur : #1}%
559     }

```

\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.

```

560 \protected\long\def\ekvd@populate@choice@#1#2%
561     {%
562     \ekvd@clear@prefixes
563     \ekvd@ifspace{#1}%
564     {\ekvd@choice@prefix{\ekv@mark#1}\ekv@mark#1\ekv@stop}%
565     {%
566     \expandafter\edef
567     \csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname
568     }%
569     {\unexpanded{#2}}%
570     }
571 \protected\def\ekvd@choice@prefix#1#2
572     {%
573     \ekv@strip{#2}{\ekvd@choice@prefix@{#1}}\ekv@mark
574     }
575 \protected\def\ekvd@choice@prefix@#1#2#3\ekv@stop
576     {%
577     \ekv@ifdefined{\ekvd@choice@p@#2}%
578     {%
579     \csname ekvd@choice@p@#2\endcsname
580     \ekvd@ifspace{#3}%
581     {\ekvd@choice@prefix{#3}#3\ekv@stop}%
582     {\ekvd@choice@prefix@done{#3}}%
583     }%
584     {\ekvd@choice@prefix@done{#1}}%
585     }
586 \protected\def\ekvd@choice@prefix@done#1%
587     {%

```



```

588     \ekvd@prot\expandafter\edef
589         \csname
590             \ekv@strip{#1}{\ekvd@choice@name\ekvd@set\ekvd@set@choice}%
591         \endcsname
592     }
593 \protected\def\ekvd@choice@p@protected{\let\ekvd@prot\protected}
594 \let\ekvd@choice@p@protect\ekvd@choice@p@protected
595 \protected\def\ekvd@choice@invalid@p#1\ekvd@ifspace#2%
596     {%
597     \expandafter\ekvd@choice@invalid@p@\expandafter{\ekv@gobble@mark#2}{#1}%
598     \ekvd@ifspace{#2}%
599     }
600 \protected\def\ekvd@choice@invalid@p@#1#2%
601     {%
602     \expandafter\ekvd@err@no@prefix@msg\expandafter{\ekvd@cur : #2 #1}{#2}%
603     }
604 \protected\def\ekvd@choice@p@long{\ekvd@choice@invalid@p{long}}%
605 \protected\def\ekvd@choice@p@also{\ekvd@choice@invalid@p{also}}%
606 \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.

```

607 \protected\long\def\ekvd@t@choice#1#2%
608     {%
609     \ekvd@ifnew{#1}%
610     {%
611     \ekvd@assert@arg
612     {%
613     \ekvd@type@choice{#1}%
614     \def\ekvd@set@choice{#1}%
615     \ekvd@populate@choice{#2}%
616     }%
617     }%
618     }

```

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

`\ekvd@t@choice-store` These two types define a special kind of choice key and are quite similar, the only difference is what the different choices do (hence they use a shared initialisation which differs in the chosen populate step).

```

619 \protected\long\expandafter\def\csname ekvd@t@choice-store\endcsname
620     {\ekvd@type@choicesspecial\ekvd@populate@choicestore}
621 \protected\long\expandafter\def\csname ekvd@t@choice-enum\endcsname
622     {\ekvd@type@choicesspecial\ekvd@populate@choiceenum}

```

`\ekvd@type@choicesspecial` Initialise similar to a choice key. The difference is that we require two arguments (which we assert), a macro to store things in, and a csv-list containing the allowed values. #1 is the populate macro according to the type used.

```

623 \protected\long\def\ekvd@type@choicesspecial#1#2#3%
624     {%
625     \ekvd@ifnew{#2}%
626     {%
627     \ekvd@assert@twoargs{#3}%
628     {%

```

```

629         \ekvd@type@choice{#2}%
630         \def\ekvd@set@choice{#2}%
631         #1#3%
632     }%
633 }%
634 }

```

We initialise the storing macro if it doesn't yet exist, and then we loop over the value list. The \edefs with \unexpanded are both necessary to be able to store macro parameter tokens (the outer protects at define time, the inner at use time).

```

\ekvd@populate@choicestore
\ekvd@populate@choicestore@

```

```

635 \protected\long\def\ekvd@populate@choicestore#1%
636 {%
637     \ekvd@newlet#1\ekv@empty
638     \ekvcsvloop{\ekvd@populate@choicestore@#1}%
639 }
640 \protected\long\def\ekvd@populate@choicestore@#1#2%
641 {%
642     \protected\expandafter\edef
643     \csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#2}\endcsname
644     {\unexpanded{\edef#1{\unexpanded{#2}}}}%
645 }

```

This is similar to the population of a choice-store type, but instead of storing the values in a macro this initialises a count and stores the position of the value in the list inside that count (zero-indexed). The space is necessary to terminate the number scanning, which is the reason we use \@firstofone (so that the space after the macro name isn't gobbled by T<sub>E</sub>X).

```

\ekvd@populate@choiceenum
\ekvd@populate@choiceenum@

```

```

646 \protected\long\def\ekvd@populate@choiceenum#1%
647 {%
648     \ekvd@newreg#1{count}%
649     \def\ekvd@tmp{0}%
650     \ekvcsvloop{\ekvd@populate@choiceenum@#1}%
651 }
652 \protected\long\def\ekvd@populate@choiceenum@#1#2%
653 {%
654     \protected\expandafter\edef
655     \csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#2}\endcsname
656     #1=\@firstofone{\ekvd@tmp} }%
657     \edef\ekvd@tmp{the\numexpr\ekvd@tmp+1\relax}%
658 }

```

(End definition for \ekvd@t@choice-store and others.)

```

\ekvd@t@unknown-choice

```

```

659 \protected\long\expandafter\def\csname ekvd@t@unknown-choice\endcsname#1#2%
660 {%
661     \ekvd@assert@new@for@name{\ekvd@unknown@choice@name\ekvd@set{#1}}%
662     {%
663         \ekvd@assert@arg
664         {%
665             \ekvd@assert@not@long
666             \ekvd@assert@not@also
667             \ekvd@prot\expandafter
668             \def\csname\ekvd@unknown@choice@name\ekvd@set{#1}\endcsname##1{#2}%

```

```

669         }%
670     }%
671 }

```

(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.

```

672 \protected\long\def\ekvd@t@unknown#1#2%
673 {%
674     \ekv@ifdefined{ekvd@type@unknown@\detokenize{#1}}%
675     {\csname ekvd@type@unknown@\detokenize{#1}\endcsname{#2}}%
676     \ekvd@err@misused@unknown
677 }

```

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

```

678 \protected\long\def\ekvd@type@unknown@noval#1%
679 {%
680     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{#1}}%
681     {%
682         \ekvd@assert@arg
683         {%
684             \ekvd@assert@not@also
685             \ekvd@assert@not@long
686             \ekvd@prot\ekvdefunknownNoVal\ekvd@set{#1}%
687         }%
688     }%
689 }

```

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.

```

690 \protected\long\def\ekvd@type@unknown@code#1%
691 {%
692     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{#1}}%
693     {%
694         \ekvd@assert@arg
695         {%
696             \ekvd@assert@not@also
697             \begingroup
698             \def\ekvd@tmp##1##2{#1}%
699             \ekv@exparg
700             {%
701                 \endgroup
702                 \ekvd@prot\ekvd@long\ekvdefunknown\ekvd@set
703             }%
704             {\ekvd@tmp{##2}{##1}}%
705         }%
706     }%
707 }

```

(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
\ekvd@type@unknown@redirect-code some prefixes.
\ekvd@type@unknown@redirect-noval
708 \protected\edef\ekvd@type@unknown@redirect#1%
709   {%
710     \expandafter\noexpand\csname ekvd@type@unknown@redirect-code\endcsname{#1}%
711     \expandafter\noexpand\csname ekvd@type@unknown@redirect-noval\endcsname{#1}%
712   }
713 \protected\expandafter\def\csname ekvd@type@unknown@redirect-code\endcsname#1%
714   {%
715     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}}u}%
716     {%
717       \ekvd@assert@arg
718       {%
719         \ekvd@assert@not@also
720         \ekvd@assert@not@protected
721         \expandafter\ekvredirectunknown\expandafter{\ekvd@set}{#1}%
722       }%
723     }%
724   }
725 \protected\expandafter\def\csname ekvd@type@unknown@redirect-noval\endcsname#1%
726   {%
727     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}}uN}%
728     {%
729       \ekvd@assert@arg
730       {%
731         \ekvd@assert@not@also
732         \ekvd@assert@not@protected
733         \ekvd@assert@not@long
734         \expandafter\ekvredirectunknownNoVal\expandafter{\ekvd@set}{#1}%
735       }%
736     }%
737   }

```

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

### 2.3.2 Key Type Helpers

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

```

\ekvd@h@choice The choice helper will just test whether the given choice was defined, if not throw an
\ekvd@h@choice@ error expandably, else call the macro which stores the code for this choice.
738 \def\ekvd@h@choice#1%
739   {%
740     \expandafter\ekvd@h@choice@
741     \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname
742     {#1}%
743   }
744 \def\ekvd@h@choice@#1#2%
745   {%
746     \ifx#1\relax
747       \ekvd@err@choice@invalid{#2}%
748       \expandafter\@gobble
749     \fi

```

```

750     #1%
751   }

```

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

### 2.3.3 Handling also

```

\ekvd@add@val
\ekvd@add@noval 752 \protected\long\def\ekvd@add@val#1#2#3%
\ekvd@add@aux    753   {%
\ekvd@add@aux@  754     \ekvd@assert@val{#1}%
                755     {%
                756       \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}\endcsname{##1}}%
                757       {#1}{#2}{\ekvd@long\ekvdef}{#3}%
                758     }%
                759   }
760 \protected\long\def\ekvd@add@noval#1#2#3%
761   {%
762     \ekvd@assert@noval{#1}%
763     {%
764       \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}\endcsname{}%
765       {#1}{#2}\ekvdefNoVal{#3}%
766     }%
767   }
768 \protected\long\def\ekvd@add@aux#1#2%
769   {%
770     \ekvd@extract@prefixes#1%
771     \expandafter\ekvd@add@aux@\expandafter{#1#2}%
772   }
773 \protected\long\def\ekvd@add@aux@#1#2#3#4#5%
774   {%
775     #5%
776     \ekvd@prot#4\ekvd@set{#2}{#1#3}%
777   }

```

(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@extract@prefixes@ \ekvd@long and \ekvd@prot accordingly.
\ekvd@extract@prefixes@long 778 \protected\def\ekvd@extract@prefixes#1%
\ekvd@extract@prefixes@prot 779   {%
                          780     \expandafter\ekvd@extract@prefixes@\meaning#1\ekvd@stop
                          781   }

```

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.

```

782 \protected\def\ekvd@extract@prefixes@#1#2#3%
783   {%
784     \protected\def\ekvd@extract@prefixes@##1#1#2\ekvd@stop
785     {%
786       \ekvd@extract@prefixes@long
787       ##1\ekvd@mark\@firstofone#2\ekvd@mark\@gobble\ekvd@stop
788       {\let\ekvd@long\long}%

```

```

789     \ekvd@extract@prefixes@prot
790     ##1\ekvd@mark\@firstofone#3\ekvd@mark\@gobble\ekvd@stop
791     {\let\ekvd@prot\protected}%
792     }%
793 \protected\def\ekvd@extract@prefixes@long##1#2##2\ekvd@mark##3##4\ekvd@stop
794 {##3}%
795 \protected\def\ekvd@extract@prefixes@prot##1#3##2\ekvd@mark##3##4\ekvd@stop
796 {##3}%
797 }

```

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`.

```

798 \begingroup
799 \edef\ekvd@tmp
800   {%
801     \endgroup
802     \ekvd@extract@prefixes@
803     {\detokenize{macro:}}%
804     {\string\long}%
805     {\string\protected}%
806   }
807 \ekvd@tmp

```

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

#### 2.3.4 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>`.

```

808 \protected\def\ekvd@newlet#1#2%
809   {%
810     \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\let#1#2}%
811   }
812 \protected\def\ekvd@newreg#1#2%
813   {%
814     \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\csname new#2\endcsname#1}%
815   }

```

*(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.

```

816 \long\def\ekvd@assert@twoargs#1%
817   {%
818     \ekvd@ifnottwoargs{#1}{\ekvd@err@missing@definition}%
819   }
820 \long\def\ekvd@ifnottwoargs#1%
821   {%
822     \ekvd@ifempty@gtwo#1\ekv@ifempty@B
823     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
824   }
825 \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.
\ekvd@assert@noval
\ekvd@assert@noval@
\ekvd@extract@args
\ekvd@extracted@args
\ekvd@one@arg@string
826 \protected\def\ekvd@assert@val#1%
827   {%
828     \ekvifdefined\ekvd@set{#1}%
829     {\expandafter\ekvd@assert@val@\csname\ekv@name\ekvd@set{#1}\endcsname}%
830     {%
831       \ekvifdefinedNoVal\ekvd@set{#1}%
832       \ekvd@err@add@val@on@noval
833       {\ekvd@err@undefined@key{#1}}%
834       \@gobble
835     }%
836   }
837 \protected\def\ekvd@assert@val@#1%
838   {%
839     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
840     \unless\ifx\ekvd@extracted@args\ekvd@one@arg@string
841       \ekvd@err@unsupported@arg
842     \fi
843     \@firstofone
844   }%
845 \protected\def\ekvd@assert@noval#1%
846   {%
847     \ekvifdefinedNoVal\ekvd@set{#1}%
848     {\expandafter\ekvd@assert@noval@\csname\ekv@name\ekvd@set{#1}N\endcsname}%
849     {%
850       \ekvifdefined\ekvd@set{#1}%
851       \ekvd@err@add@noval@on@val
852       {\ekvd@err@undefined@key{#1}}%
853       \@gobble
854     }%
855   }
856 \protected\def\ekvd@assert@noval@#1%
857   {%
858     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
859     \unless\ifx\ekvd@extracted@args\ekv@empty
860       \ekvd@err@unsupported@arg
861     \fi
862     \@firstofone
863   }
864 \protected\def\ekvd@extract@args#1%
865   {%
866     \protected\def\ekvd@extract@args##1#1##2->##3\ekvd@stop
867     {\def\ekvd@extracted@args{##2}}%
868   }
869 \expandafter\ekvd@extract@args\expandafter{\detokenize{macro:}}
870 \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

```

```

871 \def\ekvd@assert@arg{\ekvd@ifnoarg\ekvd@err@missing@definition}
872 \long\def\ekvd@assert@arg@msg#1%
873   {%
874     \ekvd@ifnoarg{\ekvd@err@missing@definition@msg{#1}}%
875   }

```

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

```

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

```

```

876 \long\def\ekvd@assert@filledarg#1%
877   {%
878     \ekvd@ifnoarg@or@empty{#1}\ekvd@err@missing@definition
879   }
880 \long\def\ekvd@ifnoarg@or@empty#1%
881   {%
882     \ekvd@ifnoarg
883     \@firstoftwo
884     {\ekv@ifempty{#1}}%
885   }

```

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

```

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

```

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

```

886 \def\ekvd@assert@not@long{\ifx\ekvd@long\long\ekvd@err@no@prefix{long}\fi}
887 \def\ekvd@assert@not@protected
888   {\ifx\ekvd@prot\protected\ekvd@err@no@prefix{protected}\fi}
889 \def\ekvd@assert@not@also{\ekvd@ifalso{\ekvd@err@no@prefix{also}}{}}
890 \def\ekvd@assert@not@long@also
891   {\ifx\ekvd@long\long\ekvd@err@no@prefix@also{long}\fi}
892 \def\ekvd@assert@not@protected@also
893   {\ifx\ekvd@prot\protected\ekvd@err@no@prefix@also{protected}\fi}
894 \def\ekvd@assert@new#1#2%
895   {\csname ekvifdefined#1\endcsname\ekvd@set{#2}{\ekvd@err@not@new}}
896 \def\ekvd@assert@not@new
897   {\ifx\ekvd@ifnew\ekvd@assert@new\ekvd@err@no@prefix{new}\fi}
898 \def\ekvd@assert@new@for@name#1%
899   {%
900     \ifx\ekvd@ifnew\ekvd@assert@new
901     \ekv@fi@firstoftwo
902     \fi
903     \@secondoftwo
904     {\ekv@ifdefined{#1}\ekvd@err@not@new}%
905     \@firstofone
906   }

```

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

```

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

```

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

```

907 \protected\def\ekvd@if@not@already@choice#1%
908   {%

```



```

909     \expandafter\ekvd@if@not@already@choice@a
910     \csname\ekv@name\ekvd@set{#1}\endcsname
911     }\ekvd@h@choice\ekvd@stop
912   }
913 \protected\def\ekvd@if@not@already@choice@a
914   {%
915     \expandafter\ekvd@if@not@already@choice@b
916   }
917 \long\protected\def\ekvd@if@not@already@choice@b#1\ekvd@h@choice#2\ekvd@stop
918   {%
919     \ekv@ifempty{#2}\@firstofone\@gobble
920   }

```

(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  
\ekvd@ifspace@ up to the first space.

```

921 \long\def\ekvd@ifspace#1%
922   {%
923     \ekvd@ifspace@#1 \ekv@ifempty@B
924     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
925   }
926 \long\def\ekvd@ifspace@#1 % keep this space
927   {%
928     \ekv@ifempty@\ekv@ifempty@A
929   }

```

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

### 2.3.5 Messages

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

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

```

\ekvd@err@missing@definition 930 \protected\def\ekvd@errm#1{\errmessage{expkv-def Error: #1}}
\ekvd@err@missing@definition@msg 931 \protected\def\ekvd@err@missing@definition
\ekvd@err@missing@type 932   {\ekvd@errm{Missing definition for key '\ekvd@cur'}}
\ekvd@err@undefined@prefix 933 \protected\def\ekvd@err@missing@definition@msg#1%
\ekvd@err@undefined@key 934   {\ekvd@errm{Missing definition for key '\ekv@unexpanded{#1}'} }
\ekvd@err@no@prefix 935 \protected\def\ekvd@err@missing@type
\ekvd@err@no@prefix@msg 936   {\ekvd@errm{Missing type prefix for key '\ekvd@cur'}}
\ekvd@err@no@prefix@also 937 \protected\def\ekvd@err@undefined@prefix#1%
\ekvd@err@add@val@on@noval 938   {%
\ekvd@err@add@noval@on@val 939     \ekvd@errm
\ekvd@err@unsupported@arg 940     {%
\ekvd@err@not@new 941       Undefined prefix '\ekv@unexpanded{#1}' found while processing
942       '\ekvd@cur'%
943     }%
944   }
945 \protected\def\ekvd@err@undefined@key#1%
946   {%
947     \ekvd@errm
948     {Undefined key '\ekv@unexpanded{#1}' found while processing '\ekvd@cur'}}%

```

```

949 }
950 \protected\def\ekvd@err@undefined@noval#1%
951   {%
952     \ekvd@errm
953     {%
954       Undefined noval key ‘\unexpanded{#1}’ found while processing
955       ‘\ekvd@cur’%
956     }%
957   }
958 \protected\def\ekvd@err@no@prefix#1%
959   {\ekvd@errm{prefix ‘#1’ not accepted in ‘\ekvd@cur’}}
960 \protected\def\ekvd@err@no@prefix@msg#1#2%
961   {\ekvd@errm{prefix ‘#2’ not accepted in ‘\ekv@unexpanded{#1}’}}
962 \protected\def\ekvd@err@no@prefix@also#1%
963   {\ekvd@errm{‘\ekvd@cur’ not allowed with a ‘#1’ key}}
964 \protected\def\ekvd@err@add@val@on@noval
965   {\ekvd@errm{‘\ekvd@cur’ not allowed with a NoVal key}}
966 \protected\def\ekvd@err@add@noval@on@val
967   {\ekvd@errm{‘\ekvd@cur’ not allowed with a value taking key}}
968 \protected\def\ekvd@err@unsupported@arg\fi@firstofone#1%
969   {%
970     \fi
971     \ekvd@errm
972     {%
973       Existing key-macro has the unsupported argument string
974       ‘\ekvd@extracted@args’ for key ‘\ekvd@cur’%
975     }%
976   }
977 \protected\def\ekvd@err@not@new
978   {\ekvd@errm{The key for ‘\ekvd@cur’ is already defined}}
979 \protected\long\def\ekvd@err@misused@unknown
980   {\ekvd@errm{Misuse of the unknown type found while processing ‘\ekvd@cur’}}

```

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

\ekvd@err@choice@invalid will have to use this mechanism to throw its message. Also we have to retrieve the name parts of the choice in an easy way, so we use parentheses of catcode 8 here, which should suffice in most cases to allow for a correct separation.

```

\ekvd@err@choice@invalid@
\ekvd@err@choice@invalid@
\ekvd@choice@name
\ekvd@unknown@choice@name
981 \def\ekvd@err@choice@invalid#1%
982   {%
983     \ekvd@err@choice@invalid@#1%
984   }
985 \begingroup
986 \catcode40=8
987 \catcode41=8
988 \@firstofone{\endgroup
989 \def\ekvd@choice@name#1#2#3%
990   {%
991     ekvd#1(#2)\detokenize{#3}%
992   }
993 \def\ekvd@unknown@choice@name#1#2%
994   {%
995     ekvd:u:#1(#2)%
996   }

```

```

997 \def\ekvd@err@choice@invalid@ ekvd#1(#2)\detokenize#3%
998   {%
999     \ekv@ifdefined{\ekvd@unknown@choice@name{#1}{#2}}%
1000     {\csname\ekvd@unknown@choice@name{#1}{#2}\endcsname{#3}}%
1001     {\ekvd@err{invalid choice ‘#3’ for ‘#2’ in set ‘#1’}}%
1002   }
1003 }

```

*(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.

```

1004 \ekv@exparg{\long\def\ekvd@err#1}{\ekv@err{expkv-def}{#1}}

```

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

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

```

1005 \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.

<b>A</b>		<code>\ekvredirectunknownNoVal</code> . . . . .	734
<code>also</code> . . . . .	3	<code>\ekvset</code> . . . . .	489
<code>apptoks</code> . . . . .	6	<code>enoval</code> . . . . .	4
<b>B</b>		<code>eskip</code> . . . . .	6
<code>bool</code> . . . . .	5	<code>estore</code> . . . . .	5
<code>boolpair</code> . . . . .	5	<b>F</b>	
<code>boolpairTF</code> . . . . .	5	<code>fdefault</code> . . . . .	4
<code>boolTF</code> . . . . .	5	<code>final</code> . . . . .	4
<code>box</code> . . . . .	6	<b>G</b>	
<b>C</b>		<code>gapptoks</code> . . . . .	6
<code>choice</code> . . . . .	7	<code>gbool</code> . . . . .	5
<code>choice-enum</code> . . . . .	7	<code>gboolpair</code> . . . . .	5
<code>choice-store</code> . . . . .	7	<code>gboolpairTF</code> . . . . .	5
<code>code</code> . . . . .	4	<code>gboolTF</code> . . . . .	5
<b>D</b>		<code>gbox</code> . . . . .	6
<code>data</code> . . . . .	5	<code>gdata</code> . . . . .	5
<code>dataT</code> . . . . .	5	<code>gdataT</code> . . . . .	5
<code>default</code> . . . . .	4	<code>gdimen</code> . . . . .	6
<code>dimen</code> . . . . .	6	<code>gint</code> . . . . .	6
<b>E</b>		<code>ginvbool</code> . . . . .	5
<code>ecode</code> . . . . .	4	<code>ginvboolTF</code> . . . . .	5
<code>edata</code> . . . . .	5	<code>gpretoks</code> . . . . .	6
<code>edataT</code> . . . . .	5	<code>gskip</code> . . . . .	6
<code>edefault</code> . . . . .	4	<code>gstore</code> . . . . .	5
<code>edimen</code> . . . . .	6	<code>gtoks</code> . . . . .	6
<code>einitial</code> . . . . .	4	<code>\gtoksapp</code> . . . . .	368
<code>eint</code> . . . . .	6	<code>\gtokspre</code> . . . . .	370
<code>\ekvchangeset</code> . . . . .	104, 107	<b>I</b>	
<code>\ekvcsvloop</code> . . . . .	638, 650	<code>initial</code> . . . . .	4
<code>\ekvdDate</code> . . . . .	2, 5, 9, 22	<code>int</code> . . . . .	6
<code>\ekvdef</code> . . . . .	292, 323, 343, 363, 386, 413, 437, 465, 546, 757	<code>invbool</code> . . . . .	5
<code>\ekvdefinekeys</code> . . . . .	2, 47	<code>invboolTF</code> . . . . .	5
<code>\ekvdefNoVal</code> . . . . .	107, 226, 255, 765	<b>L</b>	
<code>\ekvdefunknown</code> . . . . .	702	<code>long</code> . . . . .	3
<code>\ekvdefunknownNoVal</code> . . . . .	686	<b>M</b>	
<code>\ekvdVersion</code> . . . . .	2, 5, 9, 15, 22	<code>meta</code> . . . . .	6
<code>\ekvverr</code> . . . . .	1004	<b>N</b>	
<code>\ekvifdefined</code> . . . . .	152, 176, 202, 828, 850	<code>new</code> . . . . .	3
<code>\ekvifdefinedNoVal</code> . . . . .	197, 831, 847	<code>\newtoks</code> . . . . .	394
<code>\ekvlet</code> . . . . .	142, 550	<code>nmeta</code> . . . . .	6
<code>\ekvletNoVal</code> . . . . .	127, 163, 184	<code>\noexpand</code> . . . . .	710, 711
<code>\ekvparse</code> . . . . .	50, 554	<code>noval</code> . . . . .	4
<code>\ekvredirectunknown</code> . . . . .	721		

<b>O</b>	
odefault .....	4
oinitial .....	4
<b>P</b>	
pretoks .....	6
\protect .....	16
protect .....	3
protected .....	3
<b>Q</b>	
qdefault .....	4
<b>S</b>	
set .....	7
skip .....	6
smeta .....	6
snmeta .....	6
store .....	5
<b>T</b>	
TeX and L <sup>A</sup> T <sub>E</sub> X 2 <sub>ε</sub> commands:	
\ekv@empty .....	41, 42, 459, 637, 859
\ekv@exparg .....	126, 141, 162, 183, 212, 215, 482, 513, 540, 699, 1004
\ekv@expargtwice .....	103, 107, 527
\ekv@fi@firstoftwo .....	35, 901
\ekv@fi@gobble .....	810, 814
\ekv@gobble@mark .....	597
\ekv@ifdefined .....	73, 76, 577, 674, 904, 999
\ekv@ifempty .....	98, 884, 919
\ekv@ifempty@ .....	825, 928
\ekv@ifempty@A .....	823, 825, 924, 928
\ekv@ifempty@B .....	822, 823, 923, 924
\ekv@ifempty@false .....	823, 924
\ekv@mark .....	67, 70, 564, 573
\ekv@name .....	159, 181, 198, 205, 543, 680, 692, 715, 727, 756, 764, 829, 848, 910
\ekv@stop .....	67, 71, 84, 564, 575, 581
\ekv@strip .....	70, 74, 573, 590
\ekv@unexpanded .....	158, 299, 302, 305, 308, 469, 470, 934, 941, 948, 961
\ekvd@add@aux .....	542, 752
\ekvd@add@aux@ .....	752
\ekvd@add@noval .....	103, 126, 162, 183, 503, 522, 752
\ekvd@add@val .....	141, 289, 320, 341, 361, 383, 405, 435, 463, 499, 518, 752
\ekvd@arg .....	50, 52
\ekvd@assert@arg .....	121, 137, 150, 174, 611, 663, 682, 694, 717, 729, 871
\ekvd@assert@arg@msg .....	871
\ekvd@assert@filledarg .....	222, 283, 314, 335, 355, 377, 399, 429, 457, 477, 876
\ekvd@assert@new .....	91, 886
\ekvd@assert@new@for@name .....	661, 680, 692, 715, 727, 898
\ekvd@assert@not@also .....	192, 666, 684, 696, 719, 731, 889
\ekvd@assert@not@also\ekvd@assert@not@long@als .....	886
\ekvd@assert@not@long .....	94, 123, 155, 179, 193, 502, 521, 526, 665, 685, 733, 886
\ekvd@assert@not@long@also .....	503, 522, 546
\ekvd@assert@not@new .....	154, 178, 191, 886
\ekvd@assert@not@protected .....	95, 194, 720, 732, 886
\ekvd@assert@not@protected@also .....	105, 886
\ekvd@assert@noval .....	762, 826
\ekvd@assert@noval@ .....	826
\ekvd@assert@twoargs .....	266, 268, 271, 276, 509, 627, 816
\ekvd@assert@val .....	536, 754, 826
\ekvd@assert@val@ .....	826
\ekvd@choice@invalid@p .....	595, 604, 605, 606
\ekvd@choice@invalid@p@ .....	597, 600
\ekvd@choice@name .....	228, 231, 257, 260, 532, 567, 590, 643, 655, 981
\ekvd@choice@p@also .....	605
\ekvd@choice@p@long .....	524
\ekvd@choice@p@long@ .....	524
\ekvd@choice@p@new .....	606
\ekvd@choice@p@protect .....	524
\ekvd@choice@p@protected .....	524
\ekvd@choice@prefix .....	524
\ekvd@choice@prefix@ .....	524
\ekvd@choice@prefix@done .....	524
\ekvd@clear@prefixes .....	39, 64, 562
\ekvd@cur .....	65, 558, 602, 932, 936, 942, 948, 955, 959, 963, 965, 967, 974, 978, 980
\ekvd@err .....	1001, 1004
\ekvd@err@add@noval@on@val .....	851, 930
\ekvd@err@add@val@on@noval .....	832, 930
\ekvd@err@choice@invalid .....	747, 981
\ekvd@err@choice@invalid@ .....	981
\ekvd@err@missing@definition .....	99, 818, 871, 878, 930

<code>\ekvd@err@missing@definition@msg</code>	<a href="#">558, 874, 930</a>	<code>\ekvd@newreg</code>	<a href="#">316, 337, 357, 379, 401, 431, 648, 808</a>
<code>\ekvd@err@missing@type</code>	<a href="#">68, 85, 930</a>	<code>\ekvd@noarg</code>	<a href="#">50, 52</a>
<code>\ekvd@err@misused@unknown</code>	<a href="#">676, 979</a>	<code>\ekvd@one@arg@string</code>	<a href="#">826</a>
<code>\ekvd@err@no@prefix</code>	<a href="#">886, 888, 889, 897, 930</a>	<code>\ekvd@p@also</code>	<a href="#">87</a>
<code>\ekvd@err@no@prefix@also</code>	<a href="#">891, 893, 930</a>	<code>\ekvd@p@long</code>	<a href="#">87</a>
<code>\ekvd@err@no@prefix@msg</code>	<a href="#">602, 930</a>	<code>\ekvd@p@new</code>	<a href="#">87</a>
<code>\ekvd@err@not@new</code>	<a href="#">895, 904, 930</a>	<code>\ekvd@p@protect</code>	<a href="#">87</a>
<code>\ekvd@err@undefined@key</code>	<a href="#">165, 186, 208, 833, 852, 930</a>	<code>\ekvd@p@protected</code>	<a href="#">87</a>
<code>\ekvd@err@undefined@noval</code>	<a href="#">199, 950</a>	<code>\ekvd@populate@choice</code>	<a href="#">524</a>
<code>\ekvd@err@undefined@prefix</code>	<a href="#">78, 930</a>	<code>\ekvd@populate@choice@</code>	<a href="#">524</a>
<code>\ekvd@err@unsupported@arg</code>	<a href="#">841, 860, 930</a>	<code>\ekvd@populate@choice@noarg</code>	<a href="#">524</a>
<code>\ekvd@errm</code>	<a href="#">930</a>	<code>\ekvd@populate@choice@enum</code>	<a href="#">622, 646</a>
<code>\ekvd@extract@args</code>	<a href="#">826</a>	<code>\ekvd@populate@choice@enum@</code>	<a href="#">646</a>
<code>\ekvd@extract@prefixes</code>	<a href="#">770, 778</a>	<code>\ekvd@populate@choice@store</code>	<a href="#">620, 635</a>
<code>\ekvd@extract@prefixes@</code>	<a href="#">778</a>	<code>\ekvd@populate@choice@store@</code>	<a href="#">635</a>
<code>\ekvd@extract@prefixes@long</code>	<a href="#">778</a>	<code>\ekvd@prefix</code>	<a href="#">67, 70, 84</a>
<code>\ekvd@extract@prefixes@prot</code>	<a href="#">778</a>	<code>\ekvd@prefix@</code>	<a href="#">70</a>
<code>\ekvd@extracted@args</code>	<a href="#">826, 974</a>	<code>\ekvd@prefix@after@p</code>	<a href="#">77, 81</a>
<code>\ekvd@h@choice</code>	<a href="#">530, 738, 911, 917</a>	<code>\ekvd@prot</code>	<a href="#">39, 88, 124, 139, 156, 180, 288, 319, 340, 360, 382, 404, 462, 497, 527, 588, 593, 667, 686, 702, 776, 791, 888, 893</a>
<code>\ekvd@h@choice@</code>	<a href="#">738</a>	<code>\ekvd@set</code>	<a href="#">49, 107, 127, 142, 152, 159, 163, 176, 181, 184, 197, 198, 202, 205, 226, 228, 231, 255, 257, 260, 292, 323, 343, 363, 386, 413, 437, 465, 479, 483, 514, 532, 543, 550, 567, 590, 643, 655, 661, 668, 680, 686, 692, 702, 715, 721, 727, 734, 756, 764, 776, 828, 829, 831, 847, 848, 850, 895, 910</a>
<code>\ekvd@handle</code>	<a href="#">52</a>	<code>\ekvd@set@choice</code>	<a href="#">567, 590, 614, 630, 643, 655</a>
<code>\ekvd@if@not@already@choice</code>	<a href="#">538, 907</a>	<code>\ekvd@stop</code>	<a href="#">55, 60, 62, 780, 784, 787, 790, 793, 795, 839, 858, 866, 911, 917</a>
<code>\ekvd@if@not@already@choice@a</code>	<a href="#">907</a>	<code>\ekvd@t@apptoks</code>	<a href="#">349</a>
<code>\ekvd@if@not@already@choice@b</code>	<a href="#">907</a>	<code>\ekvd@t@bool</code>	<a href="#">216</a>
<code>\ekvd@ifalso</code>	<a href="#">39, 90, 101, 125, 140, 161, 182, 286, 317, 338, 358, 380, 402, 432, 460, 481, 512, 534, 889</a>	<code>\ekvd@t@boolpair</code>	<a href="#">246</a>
<code>\ekvd@ifempty@gtwo</code>	<a href="#">816</a>	<code>\ekvd@t@boolpairTF</code>	<a href="#">246</a>
<code>\ekvd@ifnew</code>	<a href="#">44, 91, 96, 119, 135, 218, 220, 248, 250, 281, 312, 333, 353, 375, 397, 427, 455, 475, 507, 609, 625, 897, 900</a>	<code>\ekvd@t@boolTF</code>	<a href="#">216</a>
<code>\ekvd@ifnoarg</code>	<a href="#">54, 59, 113, 195, 871, 882</a>	<code>\ekvd@t@box</code>	<a href="#">310</a>
<code>\ekvd@ifnoarg@or@empty</code>	<a href="#">876</a>	<code>\ekvd@t@choice</code>	<a href="#">524</a>
<code>\ekvd@ifnottwoargs</code>	<a href="#">816</a>	<code>\ekvd@t@choice-enum</code>	<a href="#">619</a>
<code>\ekvd@ifprimitive</code>	<a href="#">27, 349</a>	<code>\ekvd@t@choice-store</code>	<a href="#">619</a>
<code>\ekvd@ifspace</code>	<a href="#">66, 83, 563, 580, 595, 598, 921</a>	<code>\ekvd@t@code</code>	<a href="#">133</a>
<code>\ekvd@ifspace@</code>	<a href="#">921</a>	<code>\ekvd@t@data</code>	<a href="#">279</a>
<code>\ekvd@long</code>	<a href="#">39, 87, 139, 292, 323, 343, 363, 386, 413, 437, 465, 497, 546, 702, 757, 788, 886, 891</a>	<code>\ekvd@t@dataT</code>	<a href="#">279</a>
<code>\ekvd@mark</code>	<a href="#">787, 790, 793, 795</a>	<code>\ekvd@t@default</code>	<a href="#">148</a>
<code>\ekvd@newlet</code>	<a href="#">224, 252, 253, 285, 459, 637, 808</a>		

<code>\ekvd@t@dimen</code> .....	<a href="#">425</a>	<code>\ekvd@t@unknown-choice</code> .....	<a href="#">659</a>
<code>\ekvd@t@ecode</code> .....	<a href="#">133</a>	<code>\ekvd@t@xdata</code> .....	<a href="#">303</a>
<code>\ekvd@t@edata</code> .....	<a href="#">300</a>	<code>\ekvd@t@xdataT</code> .....	<a href="#">309</a>
<code>\ekvd@t@edataT</code> .....	<a href="#">306</a>	<code>\ekvd@t@xdimen</code> .....	<a href="#">425</a>
<code>\ekvd@t@edefault</code> .....	<a href="#">172</a>	<code>\ekvd@t@xint</code> .....	<a href="#">425</a>
<code>\ekvd@t@edimen</code> .....	<a href="#">425</a>	<code>\ekvd@t@xskip</code> .....	<a href="#">425</a>
<code>\ekvd@t@einitial</code> .....	<a href="#">189</a>	<code>\ekvd@t@xstore</code> .....	<a href="#">472</a>
<code>\ekvd@t@eint</code> .....	<a href="#">425</a>	<code>\ekvd@tmp</code> .....	
<code>\ekvd@t@enoval</code> .....	<a href="#">117</a>	... 2, 124, 126, 127, 139, 141, 142,	
<code>\ekvd@t@eskip</code> .....	<a href="#">425</a>	156, 162, 163, 180, 183, 184, 206,	
<code>\ekvd@t@estore</code> .....	<a href="#">471</a>	211, 212, 213, 215, 479, 480, 482,	
<code>\ekvd@t@fdefault</code> .....	<a href="#">148</a>	483, 497, 513, 514, 527, 545, 550,	
<code>\ekvd@t@finitial</code> .....	<a href="#">189</a>	649, 656, 657, 698, 704, 799, 807, 1005	
<code>\ekvd@t@gapptoks</code> .....	<a href="#">349</a>	<code>\ekvd@tmpa</code> .....	<a href="#">30, 34</a>
<code>\ekvd@t@gbool</code> .....	<a href="#">216</a>	<code>\ekvd@tmpb</code> .....	<a href="#">31, 34</a>
<code>\ekvd@t@gboolpair</code> .....	<a href="#">246</a>	<code>\ekvd@toks</code> .....	<a href="#">394, 407, 408, 415, 416</a>
<code>\ekvd@t@gboolpairTF</code> .....	<a href="#">246</a>	<code>\ekvd@type@apptoks</code> .....	<a href="#">373, 392, 393</a>
<code>\ekvd@t@gboolTF</code> .....	<a href="#">216</a>	<code>\ekvd@type@bool</code> .....	<a href="#">216</a>
<code>\ekvd@t@gbox</code> .....	<a href="#">310</a>	<code>\ekvd@type@boolpair</code> .....	<a href="#">246</a>
<code>\ekvd@t@gdata</code> .....	<a href="#">279</a>	<code>\ekvd@type@box</code> .....	<a href="#">310</a>
<code>\ekvd@t@gdataT</code> .....	<a href="#">279</a>	<code>\ekvd@type@choice</code> .....	<a href="#">225, 254, 524, 629</a>
<code>\ekvd@t@gdimen</code> .....	<a href="#">425</a>	<code>\ekvd@type@choicespecial</code> .....	<a href="#">620, 622, 623</a>
<code>\ekvd@t@gint</code> .....	<a href="#">425</a>	<code>\ekvd@type@code</code> .....	<a href="#">133</a>
<code>\ekvd@t@ginvbool</code> .....	<a href="#">216</a>	<code>\ekvd@type@data</code> .....	<a href="#">279</a>
<code>\ekvd@t@ginvboolTF</code> .....	<a href="#">216</a>	<code>\ekvd@type@default</code> .....	<a href="#">148</a>
<code>\ekvd@t@gpretoks</code> .....	<a href="#">370, 423</a>	<code>\ekvd@type@initial</code> .....	
<code>\ekvd@t@gskip</code> .....	<a href="#">425</a>	... 189, 211, 212, 213, 215	
<code>\ekvd@t@gstore</code> .....	<a href="#">453</a>	<code>\ekvd@type@meta</code> .....	<a href="#">473</a>
<code>\ekvd@t@gtoks</code> .....	<a href="#">331</a>	<code>\ekvd@type@meta@a</code> .....	<a href="#">473, 511</a>
<code>\ekvd@t@initial</code> .....	<a href="#">189</a>	<code>\ekvd@type@meta@b</code> .....	<a href="#">473</a>
<code>\ekvd@t@int</code> .....	<a href="#">425</a>	<code>\ekvd@type@meta@c</code> .....	<a href="#">473</a>
<code>\ekvd@t@invbool</code> .....	<a href="#">216</a>	<code>\ekvd@type@noval</code> .....	<a href="#">117</a>
<code>\ekvd@t@invboolTF</code> .....	<a href="#">216</a>	<code>\ekvd@type@preapptoks</code> .....	<a href="#">349</a>
<code>\ekvd@t@meta</code> .....	<a href="#">473</a>	<code>\ekvd@type@pretoks</code> .....	<a href="#">395, 422, 423</a>
<code>\ekvd@t@nmeta</code> .....	<a href="#">473</a>	<code>\ekvd@type@reg</code> .....	<a href="#">425</a>
<code>\ekvd@t@noval</code> .....	<a href="#">117</a>	<code>\ekvd@type@set</code> .....	<a href="#">92</a>
<code>\ekvd@t@odefault</code> .....	<a href="#">148</a>	<code>\ekvd@type@smeta</code> .....	<a href="#">505</a>
<code>\ekvd@t@oinitial</code> .....	<a href="#">189</a>	<code>\ekvd@type@smeta@</code> .....	<a href="#">505</a>
<code>\ekvd@t@pretoks</code> .....	<a href="#">369, 422</a>	<code>\ekvd@type@store</code> .....	<a href="#">453</a>
<code>\ekvd@t@qdefault</code> .....	<a href="#">148</a>	<code>\ekvd@type@toks</code> .....	<a href="#">331</a>
<code>\ekvd@t@set</code> .....	<a href="#">92</a>	<code>\ekvd@type@unknown@code</code> .....	<a href="#">672</a>
<code>\ekvd@t@skip</code> .....	<a href="#">425</a>	<code>\ekvd@type@unknown@noval</code> .....	<a href="#">672</a>
<code>\ekvd@t@smeta</code> .....	<a href="#">505</a>	<code>\ekvd@type@unknown@redirect</code> .....	<a href="#">708</a>
<code>\ekvd@t@snmeta</code> .....	<a href="#">505</a>	<code>\ekvd@type@unknown@redirect-code</code> .....	<a href="#">708</a>
<code>\ekvd@t@store</code> .....	<a href="#">453</a>	<code>\ekvd@type@unknown@redirect-noval</code> .....	<a href="#">708</a>
<code>\ekvd@t@toks</code> .....	<a href="#">331</a>	... 661, 668, 981	
<code>\ekvd@t@unknown</code> .....	<a href="#">672</a>	<code>\ekvd@prot</code> .....	<a href="#">434</a>

