

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

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

Jonathan P. Spratte\*

2021-09-20 v0.8c

## 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 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

<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	8
1.5	License	9
<b>2</b>	<b>Implementation</b>	<b>10</b>
2.1	The L <sup>A</sup> T <sub>E</sub> X Package	10
2.2	The ConT <sub>E</sub> Xt module	10
2.3	The Generic Code	10
2.3.1	Key Types	12
2.3.2	Key Type Helpers	25
2.3.3	Handling also	25
2.3.4	Tests	27
2.3.5	Messages	30
	<b>Index</b>	<b>32</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
	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 gbox	box $\langle key \rangle = \langle cs \rangle$	new also protected long
	The $\langle cs \rangle$ should be a single control sequence, such as $\backslash foo$ . Typesets $\langle value \rangle$ into a $\backslash hbox$ and stores the result in a box register. The boxes are colour safe. <b>EXP<del>R</del>VIDEF</b> doesn't provide a vbox type.	
meta	meta $\langle key \rangle = \{ \langle key \rangle = \langle value \rangle, \dots \}$	new also protected long
	This key type can set other keys, you can access the $\langle value \rangle$ which was passed to $\langle key \rangle$ inside the $\langle key \rangle = \langle value \rangle$ list with #1. It works by calling a sub- $\backslash ekvset$ on the $\langle key \rangle = \langle value \rangle$ list, so a set key will only affect that $\langle key \rangle = \langle value \rangle$ list and not the current $\backslash ekvset$ . Since it runs in a separate $\backslash ekvset$ you can't use $\backslash ekvsneak$ using keys or similar macros in the way you normally could.	
nmeta	nmeta $\langle key \rangle = \{ \langle key \rangle = \langle value \rangle, \dots \}$	new also protected long
	This key type can set other keys, the difference to meta is, that this key doesn't take a value, so the $\langle key \rangle = \langle value \rangle$ list is static.	
smeta	smeta $\langle key \rangle = \{ \langle set \rangle \} \{ \langle key \rangle = \langle value \rangle, \dots \}$	new also protected long
	Yet another meta variant. An smeta key will take a $\langle value \rangle$ which you can access using #1, but it sets the $\langle key \rangle = \langle value \rangle$ list inside of $\langle set \rangle$ , so is equal to $\backslash ekvset \{ \langle set \rangle \} \{ \langle key \rangle = \langle value \rangle, \dots \}$ .	
snmeta	snmeta $\langle key \rangle = \{ \langle set \rangle \} \{ \langle key \rangle = \langle value \rangle, \dots \}$	new also protected long
	And the last meta variant. snmeta is a combination of smeta and nmeta. It doesn't take an argument and sets the $\langle key \rangle = \langle value \rangle$ list inside of $\langle set \rangle$ .	

---

**set** `set <key> = {<set>}` new also protected long

This will define `<key>` to change the set of the current `\ekvset` invocation to `<set>`. You can omit `<set>` (including the equals sign), which is the same as using `set <key> = {<key>}`. The created `set` key will not take a `<value>`. Note that just like in `expkv` it'll not be checked whether `<set>` is defined and you'll get a low-level TeX error if you use an undefined `<set>`.

---

**choice** `choice <key> = {(value)=<definition>, ...}` new also protected long

Defines `<key>` to be a choice key, meaning it will only accept a limited set of values. You should define each possible `<value>` inside of the `<value>=<definition>` list. If a defined `<value>` is passed to `<key>` the `<definition>` will be left in the input stream. You can make individual values protected inside the `<value>=<definition>` list. By default a choice key is expandable, an undefined `<value>` will throw an error in an expandable way (but see the `unknown-choice` prefix). You can add additional choices after the `<key>` was created by using `choice` again for the same `<key>`, redefining choices is possible the same way, but there is no interface to remove certain choices.

---

**unknown-choice** `unknown-choice <key> = {<definition>}` new also protected long

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

---

**unknown\_code** `unknown code = {<definition>}` new also protected long

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

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

---

<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.

---

`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)

### 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 © 2020–2021 Jonathan P. Spratte

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

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

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

## 2 Implementation

### 2.1 The L<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{0.8c}
23 \def\ekvdDate{2021-09-20}
```

*(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@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, `ifalso` will be `\@secondoftwo`, and `ifnew` will just use its third argument.

```

27 \def\ekvd@empty{}
28 \protected\def\ekvd@clear@prefixes
29   {%
30     \let\ekvd@long\ekvd@empty
31     \let\ekvd@prot\ekvd@empty
32     \let\ekvd@ifalso\@secondoftwo
33     \long\def\ekvd@ifnew##1##2##3{##3}%
34   }
35 \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`.

```

36 \protected\def\ekvdefinekeys#1%
37   {%
38     \def\ekvd@set{#1}%
39     \ekvparse\ekvd@noarg\ekvd@arg
40   }

```

*(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@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.

```

41 \protected\long\def\ekvd@noarg#1%
42   {%
43     \let\ekvd@ifnoarg\@firstoftwo
44     \expandafter\ekvd@handle\detokenize{#1}\ekvd@stop}%
45   }
46 \protected\long\def\ekvd@arg#1%
47   {%
48     \let\ekvd@ifnoarg\@secondoftwo
49     \expandafter\ekvd@handle\detokenize{#1}\ekvd@stop
50   }
51 \protected\long\def\ekvd@handle#1\ekvd@stop#2%
52   {%
53     \ekvd@clear@prefixes
54     \edef\ekvd@cur{\detokenize{#1}}%
55     \ekvd@ifspace{#1}%
56     {\ekvd@prefix\ekv@mark#1\ekv@stop{#2}}%
57     \ekvd@err@missing@type
58   }

```

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

`\ekvd@prefix` `\ekvd@prefix@` `expkvDEF` 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).

```

59 \protected\def\ekvd@prefix#1 {\ekv@strip{#1}\ekvd@prefix@\ekv@mark}
60 \protected\def\ekvd@prefix@#1#2\ekv@stop
61   {%
62     \ekv@ifdefined{ekvd@t@#1}%
63     {\ekv@strip{#2}{\csname ekvd@t@#1\endcsname}}%
64     {%
65       \ekv@ifdefined{ekvd@p@#1}%
66       {\csname ekvd@p@#1\endcsname\ekvd@prefix@after@p{#2}}%
67       {\ekvd@err@undefined@prefix{#1}\@gobble}%
68     }%
69   }

```

(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.

```

70 \protected\def\ekvd@prefix@after@p#1%
71   {%
72     \ekvd@ifspace{#1}%
73     {\ekvd@prefix#1\ekv@stop}%
74     {\ekvd@err@missing@type\@gobble}%
75   }

```

(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`

```

76 \protected\def\ekvd@p@long{\let\ekvd@long\long}
77 \protected\def\ekvd@p@protected{\let\ekvd@prot\protected}
78 \let\ekvd@p@protect\ekvd@p@protected
79 \protected\def\ekvd@p@also{\let\ekvd@ifalso\@firstoftwo}
80 \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`

```

81 \protected\def\ekvd@type@set#1#2%
82   {%
83     \ekvd@assert@not@long
84     \ekvd@assert@not@protected
85     \ekvd@ifnew{NoVal}{#1}%
86     {%
87       \ekv@ifempty{#2}%
88       {\ekvd@err@missing@definition}%
89       {%
90         \ekvd@ifalso
91         {%

```

```

92         \ekv@expargtwice{\ekvd@add@noval{#1}}%
93         {\ekvchangeset{#2}}%
94         \ekvd@assert@not@protected@also
95     }%
96     {\ekv@expargtwice{\ekvdefNoVal\ekvd@set{#1}}{\ekvchangeset{#2}}}%
97 }%
98 }%
99 }
100 \protected\def\ekvd@t@set#1#2%
101 {%
102     \ekvd@ifnoarg
103     {\ekvd@type@set{#1}{#1}}%
104     {\ekvd@type@set{#1}{#2}}%
105 }

```

(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.

```

106 \protected\long\def\ekvd@type@noval#1#2#3%
107 {%
108     \ekvd@ifnew{NoVal}{#2}%
109     {%
110         \ekvd@assert@arg
111         {%
112             \ekvd@assert@not@long
113             \ekvd@prot#1\ekvd@tmp{#3}%
114             \ekvd@ifalso
115             {\ekv@exparg{\ekvd@add@noval{#2}}\ekvd@tmp{}}%
116             {\ekvletNoVal\ekvd@set{#2}\ekvd@tmp}%
117         }%
118     }%
119 }
120 \protected\def\ekvd@t@noval{\ekvd@type@noval\def}
121 \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 \ekvdef.  
\ekvd@t@ecode

```

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

```

```

133     }%
134   }
135   \protected\def\ekvd@t@code{\ekvd@type@code\def}
136   \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` 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.

```

137   \protected\long\def\ekvd@type@default#1#2#3#4%
138   {%
139     \ekvd@assert@arg
140     {%
141       \ekvifdefined\ekvd@set{#3}%
142       {%
143         \ekvd@assert@not@new
144         \ekvd@assert@not@long
145         \ekvd@prot\edef\ekvd@tmp
146         {%
147           \ekv@unexpanded\expandafter#1%
148           {#2\csname\ekv@name\ekvd@set{#3}\endcsname{#4}}%
149         }%
150       \ekvd@ifalso
151       {\ekv@exparg{\ekvd@add@noval{#3}}\ekvd@tmp{}}%
152       {\ekvletNoVal\ekvd@set{#3}\ekvd@tmp}%
153     }%
154     {\ekvd@err@undefined@key{#3}}%
155   }%
156 }
157 \protected\def\ekvd@t@default{\ekvd@type@default{}}
158 \protected\def\ekvd@t@qdefault{\ekvd@type@default{\expandafter\expandafter}}
159 \let\ekvd@t@odefault\ekvd@t@qdefault
160 \protected\def\ekvd@t@fdefault{\ekvd@type@default}{\romannumeral'\^^@}

```

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

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

```

161 \protected\long\def\ekvd@t@edefault#1#2%
162   {%
163     \ekvd@assert@arg
164     {%
165       \ekvifdefined\ekvd@set{#1}%
166       {%
167         \ekvd@assert@not@new
168         \ekvd@assert@not@long
169         \ekvd@prot\edef\ekvd@tmp
170         {\csname\ekv@name\ekvd@set{#1}\endcsname{#2}}%
171       \ekvd@ifalso
172       {\ekv@exparg{\ekvd@add@noval{#1}}\ekvd@tmp{}}%
173       {\ekvletNoVal\ekvd@set{#1}\ekvd@tmp}%

```

```

174     }%
175     {\ekvd@err@undefined@key{#1}}%
176   }%
177 }

```

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

```

\ekvd@t@initial
\ekvd@t@oinitial 178 \long\def\ekvd@type@initial#1#2#3#4%
\ekvd@t@finitial 179   {%
\ekvd@t@einitial 180   \ekvd@assert@not@new
181   \ekvd@assert@not@also
182   \ekvd@assert@not@long
183   \ekvd@assert@not@protected
184   \ekvd@ifnoarg
185   {%
186     \ekvifdefinedNoVal\ekvd@set{#3}%
187     {\csname\ekv@name\ekvd@set{#3}N\endcsname}%
188     {\ekvd@err@undefined@noval{#3}}%
189   }%
190   {%
191     \ekvifdefined\ekvd@set{#3}%
192     {%
193       #1{#2#4}%
194       \csname\ekv@name\ekvd@set{#3}\expandafter\endcsname\expandafter
195       {\ekvd@tmp}}%
196     }%
197     {\ekvd@err@undefined@key{#3}}%
198   }%
199 }
200 \def\ekvd@t@initial{\ekvd@type@initial{\def\ekvd@tmp}{}}
201 \def\ekvd@t@oinitial{\ekvd@type@initial{\ekv@exparg{\def\ekvd@tmp}}{}}
202 \def\ekvd@t@einitial{\ekvd@type@initial{\edef\ekvd@tmp}{}}
203 \def\ekvd@t@finitial
204   {\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 205 \protected\def\ekvd@type@bool#1#2#3#4#5%
\ekvd@t@gboolTF 206   {%
\ekvd@t@invbool 207   \ekvd@ifnew{#4}%
\ekvd@t@ginvbool 208   {%
\ekvd@t@invboolTF 209   \ekvd@ifnew{NoVal}{#4}%
\ekvd@t@ginvboolTF 210   {%
211     \ekvd@assert@filledarg{#5}%
212     {%
213       \ekvd@newlet#5#3%
214       \ekvd@type@choice{#4}%
215       \protected\ekvdefNoVal\ekvd@set{#4}{#1\let#5#2}%
216       \protected\expandafter\def
217       \csname\ekvd@choice@name\ekvd@set{#4}{true}\endcsname

```

```

218         {#1\let#5#2}%
219         \protected\expandafter\def
220         \csname\ekvd@choice@name\ekvd@set{#4}{false}\endcsname
221         {#1\let#5#3}%
222     }%
223 }%
224 }%
225 }
226 \protected\def\ekvd@t@bool{\ekvd@type@bool}\iftrue\iffalse}
227 \protected\def\ekvd@t@gbool{\ekvd@type@bool\global\iftrue\iffalse}
228 \protected\def\ekvd@t@boolTF{\ekvd@type@bool}\@firstoftwo\@secondoftwo}
229 \protected\def\ekvd@t@gboolTF{\ekvd@type@bool\global\@firstoftwo\@secondoftwo}
230 \protected\def\ekvd@t@invbool{\ekvd@type@bool}\iffalse\iftrue}
231 \protected\def\ekvd@t@ginvbool{\ekvd@type@bool\global\iffalse\iftrue}
232 \protected\def\ekvd@t@invboolTF{\ekvd@type@bool}\@secondoftwo\@firstoftwo}
233 \protected\def\ekvd@t@ginvboolTF
234 {\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
235 \protected\def\ekvd@type@boolpair#1#2#3#4#5#6%
236 {%
237     \ekvd@ifnew{#4}%
238     {%
239         \ekvd@ifnew{NoVal}{#4}%
240         {%
241             \ekvd@newlet#5#3%
242             \ekvd@newlet#6#2%
243             \ekvd@type@choice{#4}%
244             \protected\ekvdefNoVal\ekvd@set{#4}{#1\let#5#2#1\let#6#3}%
245             \protected\expandafter\def
246             \csname\ekvd@choice@name\ekvd@set{#4}{true}\endcsname
247             {#1\let#5#2#1\let#6#3}%
248             \protected\expandafter\def
249             \csname\ekvd@choice@name\ekvd@set{#4}{false}\endcsname
250             {#1\let#5#3#1\let#6#2}%
251         }%
252     }%
253 }
254 \protected\def\ekvd@t@boolpair#1#2%
255 {\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair}\iftrue\iffalse{#1}#2}}
256 \protected\def\ekvd@t@gboolpair#1#2%
257 {\ekvd@assert@twoargs{#2}{\ekvd@type@boolpair\global\iftrue\iffalse{#1}#2}}
258 \protected\def\ekvd@t@boolpairTF#1#2%
259 {%
260     \ekvd@assert@twoargs{#2}%
261     {\ekvd@type@boolpair}\@firstoftwo\@secondoftwo{#1}#2}%
262 }
263 \protected\def\ekvd@t@gboolpairTF#1#2%
264 {%
265     \ekvd@assert@twoargs{#2}%
266     {\ekvd@type@boolpair\global\@firstoftwo\@secondoftwo{#1}#2}%
267 }

```



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

```
\ekvd@type@data
  \ekvd@t@data 268 \protected\def\ekvd@type@data#1#2#3#4#5#6%
  \ekvd@t@gdata 269 {%
  \ekvd@t@dataT 270 \ekvd@ifnew{#5}%
  \ekvd@t@gdataT 271 {%
    272 \ekvd@assert@filledarg{#6}%
    273 {%
    274 \ekvd@newlet#6#1%
    275 \ekvd@ifalso
    276 {%
    277 \let\ekvd@prot\protected
    278 \ekvd@add@val{#5}{\long#2#6###1#3{###1{#4}}}{}%
    279 }%
    280 {%
    281 \protected\ekvd@long\ekvdef\ekvd@set{#5}%
    282 {\long#2#6###1#3{###1{#4}}}%
    283 }%
    284 }%
    285 }%
  286 }
  287 \protected\def\ekvd@t@data
  288 {\ekvd@type@data\@secondoftwo\edef{###2}{\ekv@unexpanded{##1}}}
  289 \protected\def\ekvd@t@edata{\ekvd@type@data\@secondoftwo\edef{###2}{##1}}
  290 \protected\def\ekvd@t@gdata
  291 {\ekvd@type@data\@secondoftwo\xdef{###2}{\ekv@unexpanded{##1}}}
  292 \protected\def\ekvd@t@xdata{\ekvd@type@data\@secondoftwo\xdef{###2}{##1}}
  293 \protected\def\ekvd@t@dataT
  294 {\ekvd@type@data\@gobble\edef{#1}{\ekv@unexpanded{##1}}}
  295 \protected\def\ekvd@t@edataT{\ekvd@type@data\@gobble\edef{#1}{##1}}
  296 \protected\def\ekvd@t@gdataT
  297 {\ekvd@type@data\@gobble\xdef{#1}{\ekv@unexpanded{##1}}}
  298 \protected\def\ekvd@t@xdataT{\ekvd@type@data\@gobble\xdef{#1}{##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

```
299 \protected\def\ekvd@type@box#1#2#3%
300 {%
301 \ekvd@ifnew{#2}%
302 {%
303 \ekvd@assert@filledarg{#3}%
304 {%
305 \ekvd@newreg#3{box}%
306 \ekvd@ifalso
307 {%
308 \let\ekvd@prot\protected
309 \ekvd@add@val{#2}{#1\setbox#3\hbox{\begingroup##1\endgroup}}{}%
310 }%
311 }%
```

```

312             \protected\ekvd@long\ekvdef\ekvd@set{#2}%
313             {#1\setbox#3\hbox{\begingroup##1\endgroup}}%
314         }%
315     }%
316 }%
317 }
318 \protected\def\ekvd@t@box{\ekvd@type@box{}}
319 \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
320 \protected\def\ekvd@type@toks#1#2#3%
321     {%
322     \ekvd@ifnew{#2}%
323     {%
324     \ekvd@assert@filledarg{#3}%
325     {%
326     \ekvd@newreg#3{toks}%
327     \ekvd@ifalso
328     {%
329     \let\ekvd@prot\protected
330     \ekvd@add@val{#2}{#1#3{##1}}{}}%
331     }%
332     {\protected\ekvd@long\ekvdef\ekvd@set{#2}{#1#3{##1}}}%
333     }%
334     }%
335     }
336 \protected\def\ekvd@t@toks{\ekvd@type@toks{}}
337 \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
338 \protected\def\ekvd@type@apptoks#1#2#3%
339     {%
340     \ekvd@ifnew{#2}%
341     {%
342     \ekvd@assert@filledarg{#3}%
343     {%
344     \ekvd@newreg#3{toks}%
345     \ekvd@ifalso
346     {%
347     \let\ekvd@prot\protected
348     \ekvd@add@val{#2}{#1#3\expandafter{\the#3##1}}{}}%
349     }%
350     {%
351     \protected\ekvd@long\ekvdef\ekvd@set{#2}%
352     {#1#3\expandafter{\the#3##1}}%
353     }%
354     }%
355     }%
356     }

```

```

357 \protected\def\ekvd@t@apptoks{\ekvd@type@apptoks{}}
358 \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
359 \protected\def\ekvd@type@reg#1#2#3#4#5#6%
360   {%
361     \ekvd@ifnew{#5}%
362     {%
363       \ekvd@assert@filledarg{#6}%
364       {%
365         \ekvd@newreg#6{#1}%
366         \ekvd@ifalso
367         {%
368           \let\ekvd@prot\protected
369           \ekvd@add@val{#5}{#2#6=#3##1#4\relax}{}%
370         }%
371         {\protected\ekvd@long\ekvdef\ekvd@set{#5}{#2#6=#3##1#4\relax}}%
372       }%
373     }%
374   }
375 \protected\def\ekvd@t@int{\ekvd@type@reg{count}{-}{-}}
376 \protected\def\ekvd@t@eint{\ekvd@type@reg{count}{-}\numexpr\relax}
377 \protected\def\ekvd@t@gint{\ekvd@type@reg{count}\global{-}}
378 \protected\def\ekvd@t@xint{\ekvd@type@reg{count}\global\numexpr\relax}
379 \protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{-}{-}}
380 \protected\def\ekvd@t@edimen{\ekvd@type@reg{dimen}{-}\dimexpr\relax}
381 \protected\def\ekvd@t@gdimen{\ekvd@type@reg{dimen}\global{-}}
382 \protected\def\ekvd@t@xdimen{\ekvd@type@reg{dimen}\global\dimexpr\relax}
383 \protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{-}{-}}
384 \protected\def\ekvd@t@eskip{\ekvd@type@reg{skip}{-}\glueexpr\relax}
385 \protected\def\ekvd@t@gskip{\ekvd@type@reg{skip}\global{-}}
386 \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
387 \protected\def\ekvd@type@store#1#2#3#4%
388   {%
389     \ekvd@ifnew{#3}%
390     {%
391       \ekvd@assert@filledarg{#4}%
392       {%
393         \ekvd@newlet#4\ekvd@empty
394         \ekvd@ifalso
395         {%
396           \let\ekvd@prot\protected
397           \ekvd@add@val{#3}{#1#4#2}{}%
398         }%
399         {\protected\ekvd@long\ekvdef\ekvd@set{#3}{#1#4#2}}%
400       }%

```

```

401     }%
402   }
403   \protected\def\ekvd@t@store{\ekvd@type@store\edef{\ekv@unexpanded{##1}}}
404   \protected\def\ekvd@t@gstore{\ekvd@type@store\xdef{\ekv@unexpanded{##1}}}
405   \protected\def\ekvd@t@estore{\ekvd@type@store\edef{##1}}
406   \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>`.

```

407   \protected\long\def\ekvd@type@meta#1#2#3#4#5#6#7%
408     {%
409       \ekvd@ifnew{#1}{#6}%
410       {%
411         \ekvd@assert@filledarg{#7}%
412         {%
413           \edef\ekvd@tmp{\ekvd@set}%
414           \expandafter\ekvd@type@meta@a\expandafter{\ekvd@tmp}{#7}{#2}%
415           \ekvd@ifalso
416             {\ekv@exparg{#3{#6}}{\ekvd@tmp#4}{#5}}%
417             {\csname ekvlet#1\endcsname\ekvd@set{#6}\ekvd@tmp}%
418         }%
419       }%
420     }
421   \protected\long\def\ekvd@type@meta@a#1#2%
422     {%
423       \expandafter\ekvd@type@meta@b\expandafter{\ekvset{#1}{#2}}%
424     }
425   \protected\def\ekvd@type@meta@b
426     {%
427       \expandafter\ekvd@type@meta@c\expandafter
428     }
429   \protected\long\def\ekvd@type@meta@c#1#2%
430     {%
431       \ekvd@prot\ekvd@long\def\ekvd@tmp#2{#1}%
432     }
433   \protected\def\ekvd@t@meta{\ekvd@type@meta}{##1}\ekvd@add@val{{##1}}{}
434   \protected\def\ekvd@t@nmeta
435     {%
436       \ekvd@assert@not@long
437       \ekvd@type@meta{NoVal}{}\ekvd@add@noval{}\ekvd@assert@not@long@also
438     }

```

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

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

```

439   \protected\long\def\ekvd@type@smeta#1#2#3#4#5#6#7%
440     {%
441       \ekvd@ifnew{#1}{#6}%
442       {%
443         \ekvd@assert@twoargs{#7}%
444       }%

```

```

445         \ekvd@type@meta@a#7{#2}%
446         \ekvd@ifalso
447             {\ekv@exparg{#3{#6}}{\ekvd@tmp#4}{#5}}%
448             {\csname ekvlet#1\endcsname\ekvd@set{#6}\ekvd@tmp}%
449         }%
450     }%
451 }
452 \protected\def\ekvd@t@smeta{\ekvd@type@smeta}{##1}\ekvd@add@val{##1}{}}
453 \protected\def\ekvd@t@snmeta
454     {%
455     \ekvd@assert@not@long
456     \ekvd@type@smeta{NoVal}{}\ekvd@add@noval{}\ekvd@assert@not@long@also
457     }

```

(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@populate@choice
\ekvd@populate@choice@
\ekvd@populate@choice@noarg
\ekvd@choice@prefix
\ekvd@choice@prefix@
\ekvd@choice@p@protected
\ekvd@choice@p@protect
\ekvd@choice@p@long
\ekvd@choice@p@long@
\ekvd@t@choice
458 \protected\def\ekvd@type@choice#1%
459     {%
460     \ekvd@assert@not@long
461     \ekvd@prot\edef\ekvd@tmp##1%
462     {\ekv@unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#1}{##1}}}%
463     \ekvd@ifalso
464     {%
465     \ekvd@assert@val{#1}%
466     {%
467     \ekvd@if@not@already@choice{#1}%
468     {%
469     \ekv@exparg
470     {%
471     \expandafter\ekvd@add@aux
472     \csname\ekv@name\ekvd@set{#1}\endcsname{##1}{#1}%
473     }%
474     {\ekvd@tmp{##1}}%
475     {\ekvd@long\ekvdef}\ekvd@assert@not@long@also
476     }%
477     }%
478     }%
479     {\ekvlet\ekvd@set{#1}\ekvd@tmp}%
480     }

```

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

```

481 \protected\def\ekvd@populate@choice
482     {%
483     \ekvparse\ekvd@populate@choice@noarg\ekvd@populate@choice@
484     }
485 \protected\long\def\ekvd@populate@choice@noarg#1%
486     {%
487     \expandafter\ekvd@err@missing@definition@msg\expandafter{\ekvd@cur : #1}%
488     }

```

\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.

```

489 \protected\long\def\ekvd@populate@choice@#1#2%
490   {%
491     \ekvd@clear@prefixes
492     \expandafter\ekvd@assert@arg@msg\expandafter{\ekvd@cur : #1}%
493     {%
494       \ekvd@ifspace{#1}%
495       {\ekvd@choice@prefix\ekv@mark#1\ekv@stop}%
496       {%
497         \expandafter\def
498         \csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname
499         }%
500       {#2}%
501     }%
502   }
503 \protected\def\ekvd@choice@prefix#1
504   {%
505     \ekv@strip{#1}\ekvd@choice@prefix@\ekv@mark
506   }
507 \protected\def\ekvd@choice@prefix@#1#2\ekv@stop
508   {%
509     \ekv@ifdefined{ekvd@choice@p@#1}%
510     {%
511       \csname ekvd@choice@p@#1\endcsname
512       \ekvd@ifspace{#2}%
513       {\ekvd@choice@prefix#2\ekv@stop}%
514       {%
515         \ekvd@prot\expandafter\def
516         \csname
517         \ekv@strip{#2}\ekvd@choice@name\ekvd@set\ekvd@set@choice}%
518         \endcsname
519       }%
520     }%
521     {\ekvd@err@undefined@prefix{#1}\@gobble}%
522   }
523 \protected\def\ekvd@choice@p@protected{\let\ekvd@prot\protected}
524 \let\ekvd@choice@p@protect\ekvd@choice@p@protected
525 \protected\def\ekvd@choice@invalid@p#1\ekvd@ifspace#2%
526   {%
527     \expandafter\ekvd@choice@invalid@p@\expandafter{\ekv@gobble@mark#2}{#1}%
528     \ekvd@ifspace{#2}%
529   }
530 \protected\def\ekvd@choice@invalid@p@#1#2%
531   {%
532     \expandafter\ekvd@err@no@prefix@msg\expandafter{\ekvd@cur : #2 #1}{#2}%
533   }
534 \protected\def\ekvd@choice@p@long{\ekvd@choice@invalid@p{long}}%
535 \protected\def\ekvd@choice@p@also{\ekvd@choice@invalid@p{also}}%
536 \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.

```

537 \protected\long\def\ekvd@t@choice#1#2%
538   {%
539     \ekvd@ifnew{#1}%
540     {%
541       \ekvd@assert@arg
542       {%
543         \ekvd@type@choice{#1}%
544         \def\ekvd@set@choice{#1}%
545         \ekvd@populate@choice{#2}%
546       }%
547     }%
548   }

```

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

\ekvd@t@unknown-choice

```

549 \protected\long\expandafter\def\csname ekvd@t@unknown-choice\endcsname#1#2%
550   {%
551     \ekvd@assert@new@for@name{\ekvd@unknown@choice@name\ekvd@set{#1}}%
552     {%
553       \ekvd@assert@arg
554       {%
555         \ekvd@assert@not@long
556         \ekvd@assert@not@also
557         \ekvd@prot\expandafter
558         \def\csname\ekvd@unknown@choice@name\ekvd@set{#1}\endcsname##1{#2}%
559       }%
560     }%
561   }

```

(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.

```

562 \protected\long\def\ekvd@t@unknown#1#2%
563   {%
564     \ekv@ifdefined{ekvd@type@unknown@\detokenize{#1}}%
565     {\csname ekvd@type@unknown@\detokenize{#1}\endcsname{#2}}%
566     \ekvd@err@misused@unknown
567   }

```

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

```

568 \protected\long\def\ekvd@type@unknown@noval#1%
569   {%
570     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{#1}}%
571     {%
572       \ekvd@assert@arg
573       {%
574         \ekvd@assert@not@also
575         \ekvd@assert@not@long
576         \ekvd@prot\ekvdefunknownNoVal\ekvd@set{#1}%

```

```

577     }%
578   }%
579 }

```

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.

```

580 \protected\long\def\ekvd@type@unknown@code#1%
581   {%
582     \ekvd@assert@new@for@name{\ekv@name\ekvd@set-{}u}%
583     {%
584       \ekvd@assert@arg
585       {%
586         \ekvd@assert@not@also
587         \begingroup
588         \def\ekvd@tmp##1##2{#1}%
589         \ekv@exparg
590         {%
591           \endgroup
592           \ekvd@prot\ekvd@long\ekvdefunknown\ekvd@set
593         }%
594         {\ekvd@tmp{##2}{##1}}%
595       }%
596     }%
597 }

```

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

```

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

```

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

```

598 \protected\edef\ekvd@type@unknown@redirect#1%
599   {%
600     \expandafter\noexpand\csname ekvd@type@unknown@redirect-code\endcsname{#1}%
601     \expandafter\noexpand\csname ekvd@type@unknown@redirect-noval\endcsname{#1}%
602   }
603 \protected\expandafter\def\csname ekvd@type@unknown@redirect-code\endcsname#1%
604   {%
605     \ekvd@assert@new@for@name{\ekv@name\ekvd@set-{}u}%
606     {%
607       \ekvd@assert@arg
608       {%
609         \ekvd@assert@not@also
610         \ekvd@assert@not@protected
611         \expandafter\ekvredirectunknown\expandafter{\ekvd@set-{}#1}%
612       }%
613     }%
614   }
615 \protected\expandafter\def\csname ekvd@type@unknown@redirect-noval\endcsname#1%
616   {%
617     \ekvd@assert@new@for@name{\ekv@name\ekvd@set-{}uN}%
618     {%
619       \ekvd@assert@arg
620       {%
621         \ekvd@assert@not@also

```



```

622         \ekvd@assert@not@protected
623         \ekvd@assert@not@long
624         \expandafter\ekvredirectunknownNoVal\expandafter{\ekvd@set}{#1}%
625     }%
626 }%
627 }

```

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

### 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 error expandably, else call the macro which stores the code for this choice.

```

628 \def\ekvd@h@choice#1%
629   {%
630     \expandafter\ekvd@h@choice@
631     \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname
632     {#1}%
633   }
634 \def\ekvd@h@choice@#1#2%
635   {%
636     \ifx#1\relax
637       \ekvd@err@choice@invalid{#2}%
638       \expandafter\@gobble
639     \fi
640     #1%
641   }

```

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

### 2.3.3 Handling also

```

\ekvd@add@val
\ekvd@add@noval
\ekvd@add@aux
\ekvd@add@aux@
642 \protected\long\def\ekvd@add@val#1#2#3%
643   {%
644     \ekvd@assert@val{#1}%
645     {%
646       \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}\endcsname{{#1}}%
647       {#1}{#2}{\ekvd@long\ekvdef}{#3}%
648     }%
649   }
650 \protected\long\def\ekvd@add@noval#1#2#3%
651   {%
652     \ekvd@assert@noval{#1}%
653     {%
654       \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}N\endcsname{}%
655       {#1}{#2}\ekvdefNoVal{#3}%
656     }%
657   }
658 \protected\long\def\ekvd@add@aux#1#2%
659   {%

```

```

660 \ekvd@extract@prefixes#1%
661 \expandafter\ekvd@add@aux@\expandafter{#1#2}%
662 }
663 \protected\long\def\ekvd@add@aux@#1#2#3#4#5%
664 {%
665 #5%
666 \ekvd@prot#4\ekvd@set{#2}-{#1#3}%
667 }

```

(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.

```

668 \protected\def\ekvd@extract@prefixes#1%
669 {%
670 \expandafter\ekvd@extract@prefixes@\meaning#1\ekvd@stop
671 }

```

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.

```

672 \protected\def\ekvd@extract@prefixes@#1#2#3%
673 {%
674 \protected\def\ekvd@extract@prefixes@##1#1##2\ekvd@stop
675 {%
676 \ekvd@extract@prefixes@long
677 ##1\ekvd@mark\@firstofone#2\ekvd@mark\@gobble\ekvd@stop
678 {\let\ekvd@long\long}%
679 \ekvd@extract@prefixes@prot
680 ##1\ekvd@mark\@firstofone#3\ekvd@mark\@gobble\ekvd@stop
681 {\let\ekvd@prot\protected}%
682 }%
683 \protected\def\ekvd@extract@prefixes@long##1#2##2\ekvd@mark##3##4\ekvd@stop
684 {##3}%
685 \protected\def\ekvd@extract@prefixes@prot##1#3##2\ekvd@mark##3##4\ekvd@stop
686 {##3}%
687 }

```

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.

```

688 \begingroup
689 \edef\ekvd@tmp
690 {%
691 \endgroup
692 \ekvd@extract@prefixes@
693 {\detokenize{macro:}}%
694 {\string\long}%
695 {\string\protected}%
696 }
697 \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⟩`.

```

698 \protected\def\ekvd@newlet#1#2%
699   {%
700     \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\let#1#2}%
701   }
702 \protected\def\ekvd@newreg#1#2%
703   {%
704     \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\csname new#2\endcsname#1}%
705   }

```

*(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.

```

706 \long\def\ekvd@assert@twoargs#1%
707   {%
708     \ekvd@ifnottwoargs{#1}{\ekvd@err@missing@definition}%
709   }
710 \long\def\ekvd@ifnottwoargs#1%
711   {%
712     \ekvd@ifempty@gtwo#1\ekv@ifempty@B
713     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
714   }
715 \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.

```

716 \protected\def\ekvd@assert@val#1%
717   {%
718     \ekvifdefined\ekvd@set{#1}%
719     {\expandafter\ekvd@assert@val@\csname\ekv@name\ekvd@set{#1}\endcsname}%
720     {%
721       \ekvifdefinedNoVal\ekvd@set{#1}%
722       \ekvd@err@add@val@on@noval
723       {\ekvd@err@undefined@key{#1}}%
724       \@gobble
725     }%
726   }
727 \protected\def\ekvd@assert@val@#1%
728   {%
729     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
730     \unless\ifx\ekvd@extracted@args\ekvd@one@arg@string
731     \ekvd@err@unsupported@arg
732     \fi
733     \@firstofone
734   }%
735 \protected\def\ekvd@assert@noval#1%

```

```

736   {%
737     \ekvifdefinedNoVal\ekvd@set{#1}%
738     {\expandafter\ekvd@assert@noval@\csname\ekv@name\ekvd@set{#1}N\endcsname}%
739     {%
740       \ekvifdefined\ekvd@set{#1}%
741       \ekvd@err@add@noval@on@val
742       {\ekvd@err@undefined@key{#1}}%
743       \@gobble
744     }%
745   }
746 \protected\def\ekvd@assert@noval@#1%
747   {%
748     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
749     \unless\ifx\ekvd@extracted@args\ekvd@empty
750     \ekvd@err@unsupported@arg
751     \fi
752     \@firstofone
753   }
754 \protected\def\ekvd@extract@args#1%
755   {%
756     \protected\def\ekvd@extract@args##1##2->##3\ekvd@stop
757     {\def\ekvd@extracted@args{##2}}%
758   }
759 \expandafter\ekvd@extract@args\expandafter{\detokenize{macro:}}
760 \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
761 \def\ekvd@assert@arg{\ekvd@ifnoarg\ekvd@err@missing@definition}
762 \long\def\ekvd@assert@arg@msg#1%
763   {%
764     \ekvd@ifnoarg{\ekvd@err@missing@definition@msg{#1}}%
765   }

```

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

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

```

766 \long\def\ekvd@assert@filledarg#1%
767   {%
768     \ekvd@ifnoarg@or@empty{#1}\ekvd@err@missing@definition
769   }
770 \long\def\ekvd@ifnoarg@or@empty#1%
771   {%
772     \ekvd@ifnoarg
773     \@firstoftwo
774     {\ekv@ifempty{#1}}%
775   }

```

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

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

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

```

776 \def\ekvd@assert@not@long{\ifx\ekvd@long\long\ekvd@err@no@prefix{long}\fi}
777 \def\ekvd@assert@not@protected
778   {\ifx\ekvd@prot\protected\ekvd@err@no@prefix{protected}\fi}
779 \def\ekvd@assert@not@also{\ekvd@ifalso{\ekvd@err@no@prefix{also}}{}}
780 \def\ekvd@assert@not@long@also
781   {\ifx\ekvd@long\long\ekvd@err@no@prefix@also{long}\fi}
782 \def\ekvd@assert@not@protected@also
783   {\ifx\ekvd@prot\protected\ekvd@err@no@prefix@also{protected}\fi}
784 \def\ekvd@assert@new#1#2%
785   {\csname ekvifdefined#1\endcsname\ekvd@set{#2}{\ekvd@err@not@new}}
786 \def\ekvd@assert@not@new
787   {\ifx\ekvd@ifnew\ekvd@assert@new\ekvd@err@no@prefix{new}\fi}
788 \def\ekvd@assert@new@for@name#1%
789   {%
790     \ifx\ekvd@ifnew\ekvd@assert@new
791       \ekv@fi@firstoftwo
792     \fi
793     \@secondoftwo
794     {\ekv@ifdefined{#1}\ekvd@err@not@new}%
795     \@firstofone
796   }

```

(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.

```

797 \protected\def\ekvd@if@not@already@choice#1%
798   {%
799     \expandafter\ekvd@if@not@already@choice@a
800     \csname\ekv@name\ekvd@set{#1}\endcsname
801     }\ekvd@h@choice\ekvd@stop
802   }
803 \protected\def\ekvd@if@not@already@choice@a
804   {%
805     \expandafter\ekvd@if@not@already@choice@b
806   }
807 \long\protected\def\ekvd@if@not@already@choice@b#1\ekvd@h@choice#2\ekvd@stop
808   {%
809     \ekv@ifempty{#2}\@firstofone@gobble
810   }

```

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

```

\ekvd@ifspace
\ekvd@ifspace@

```

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

```

811 \long\def\ekvd@ifspace#1%
812   {%
813     \ekvd@ifspace@#1 \ekv@ifempty@B
814     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
815   }
816 \long\def\ekvd@ifspace@#1 % keep this space
817   {%
818     \ekv@ifempty@\ekv@ifempty@A
819   }

```

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

### 2.3.5 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 820 \protected\def\ekvd@errm#1{\errmessage{expkv-def Error: #1}}
\ekvd@err@missing@definition@msg 821 \protected\def\ekvd@err@missing@definition
\ekvd@err@missing@type 822 {\ekvd@errm{Missing definition for key '\ekvd@cur'}}
\ekvd@err@undefined@prefix 823 \protected\def\ekvd@err@missing@definition@msg#1%
\ekvd@err@undefined@key 824 {\ekvd@errm{Missing definition for key '\ekv@unexpanded{#1}'}}
\ekvd@err@no@prefix 825 \protected\def\ekvd@err@missing@type
\ekvd@err@no@prefix@msg 826 {\ekvd@errm{Missing type prefix for key '\ekvd@cur'}}
\ekvd@err@no@prefix@also 827 \protected\def\ekvd@err@undefined@prefix#1%
\ekvd@err@add@val@on@noval 828 {%
\ekvd@err@add@noval@on@val 829 \ekvd@errm
\ekvd@err@unsupported@arg 830 {%
\ekvd@err@not@new 831 Undefined prefix '\ekv@unexpanded{#1}' found while processing
832 '\ekvd@cur'%
833 }%
834 }
835 \protected\def\ekvd@err@undefined@key#1%
836 {%
837 \ekvd@errm
838 {Undefined key '\ekv@unexpanded{#1}' found while processing '\ekvd@cur'}%
839 }
840 \protected\def\ekvd@err@undefined@noval#1%
841 {%
842 \ekvd@errm
843 {%
844 Undefined noval key '\unexpanded{#1}' found while processing
845 '\ekvd@cur'%
846 }%
847 }
848 \protected\def\ekvd@err@no@prefix#1%
849 {\ekvd@errm{prefix '#1' not accepted in '\ekvd@cur'}}
850 \protected\def\ekvd@err@no@prefix@msg#1#2%
851 {\ekvd@errm{prefix '#2' not accepted in '\ekv@unexpanded{#1}'}}
852 \protected\def\ekvd@err@no@prefix@also#1%
853 {\ekvd@errm{'\ekvd@cur' not allowed with a '#1' key}}
854 \protected\def\ekvd@err@add@val@on@noval
855 {\ekvd@errm{'\ekvd@cur' not allowed with a NoVal key}}
856 \protected\def\ekvd@err@add@noval@on@val
857 {\ekvd@errm{'\ekvd@cur' not allowed with a value taking key}}
858 \protected\def\ekvd@err@unsupported@arg\fi\@firstofone#1%
859 {%
860 \fi
861 \ekvd@errm
862 {%
863 Existing key-macro has the unsupported argument string
864 '\ekvd@extracted@args' for key '\ekvd@cur'%
865 }%

```

```

866 }
867 \protected\def\ekvd@err@not@new
868   {\ekvd@errm{The key for ‘\ekvd@cur’ is already defined}}
869 \protected\long\def\ekvd@err@misused@unknown
870   {\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
871 \def\ekvd@err@choice@invalid#1%
872   {%
873     \ekvd@err@choice@invalid@#1\ekv@stop
874   }
875 \begingroup
876 \catcode40=8
877 \catcode41=8
878 \@firstofone{\endgroup
879 \def\ekvd@choice@name#1#2#3%
880   {%
881     ekvd#1(#2)#3%
882   }
883 \def\ekvd@unknown@choice@name#1#2%
884   {%
885     ekvd:u:#1(#2)%
886   }
887 \def\ekvd@err@choice@invalid@ ekvd#1(#2)#3\ekv@stop%
888   {%
889     \ekv@ifdefined{\ekvd@unknown@choice@name{#1}{#2}}%
890     {\csname\ekvd@unknown@choice@name{#1}{#2}\endcsname{#3}}%
891     {\ekvd@err{invalid choice ‘#3’ for ‘#2’ in set ‘#1’}}%
892   }
893 }

```

(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.

```

894 \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 @.

```

895 \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>		estore .....	5
also .....	3	<b>F</b>	
apptoks .....	6	fdefault .....	4
<b>B</b>		finitial .....	4
bool .....	5	<b>G</b>	
boolpair .....	5	gapptoks .....	6
boolpairTF .....	5	gbool .....	5
boolTF .....	5	gboolpair .....	5
box .....	6	gboolpairTF .....	5
<b>C</b>		gboolTF .....	5
choice .....	7	gbox .....	6
code .....	4	gdata .....	5
<b>D</b>		gdataT .....	5
data .....	5	gdimen .....	6
dataT .....	5	gint .....	6
default .....	4	ginvbool .....	5
dimen .....	6	ginvboolTF .....	5
<b>E</b>		gskip .....	6
ecode .....	4	gstore .....	5
edata .....	5	gtoks .....	6
edataT .....	5	<b>I</b>	
edefault .....	4	initial .....	4
edimen .....	6	int .....	6
einitial .....	4	invbool .....	5
eint .....	6	invboolTF .....	5
\ekvchangeset .....	93, 96	<b>L</b>	
\ekvdDate .....	2, 5, 9, <u>22</u>	long .....	3
\ekvdef 281, 312, 332, 351, 371, 399, 475, 647		<b>M</b>	
\ekvdefinekeys .....	2, <u>36</u>	meta .....	6
\ekvdefNoVal .....	96, 215, 244, 655	<b>N</b>	
\ekvdefunknown .....	592	new .....	3
\ekvdefunknownNoVal .....	576	nmeta .....	6
\ekvdVersion .....	2, 5, 9, 15, <u>22</u>	\noexpand .....	600, 601
\ekvverr .....	894	noval .....	4
\ekvifdefined .....	141, 165, 191, 718, 740	<b>O</b>	
\ekvifdefinedNoVal .....	186, 721, 737	odefault .....	4
\ekvlet .....	131, 479	oinitial .....	4
\ekvletNoVal .....	116, 152, 173	<b>P</b>	
\ekvparse .....	39, 483	\protect .....	16
\ekvredirectunknown .....	611	protect .....	3
\ekvredirectunknownNoVal .....	624	protected .....	3
\ekvset .....	423		
enoval .....	4		
eskip .....	6		



	<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@exparg	.....	115, 130, 151, 172, 201, 204, 416, 447, 469, 589, 894
\ekv@expargtwice	.....	92, 96
\ekv@fi@firstoftwo	.....	791
\ekv@fi@gobble	.....	700, 704
\ekv@gobble@mark	.....	527
\ekv@ifdefined	62, 65, 509, 564, 794, 889	
\ekv@ifempty	.....	87, 774, 809
\ekv@ifempty@	.....	715, 818
\ekv@ifempty@A	....	713, 715, 814, 818
\ekv@ifempty@B	....	712, 713, 813, 814
\ekv@ifempty@false	.....	713, 814
\ekv@mark	.....	56, 59, 495, 505
\ekv@name	148, 170, 187, 194, 472, 570, 582, 605, 617, 646, 654, 719, 738, 800	
\ekv@stop	.....	56, 60, 73, 495, 507, 513, 873, 887
\ekv@strip	.....	59, 63, 505, 517
\ekv@unexpanded	147, 288, 291, 294, 297, 403, 404, 462, 824, 831, 838, 851	
\ekvd@add@aux	.....	471, <u>642</u>
\ekvd@add@aux@	.....	<u>642</u>
\ekvd@add@noval	.....	92, 115, 151, 172, 437, 456, <u>642</u>
\ekvd@add@val	.....	130, 278, 309, 330, 348, 369, 397, 433, 452, <u>642</u>
\ekvd@arg	.....	39, <u>41</u>
\ekvd@assert@arg	... 110, 126, 139, 163, 541, 553, 572, 584, 607, 619, <u>761</u>	
\ekvd@assert@arg@msg	.....	492, <u>761</u>
\ekvd@assert@filledarg	.....	211, 272, 303, 324, 342, 363, 391, 411, <u>766</u>
\ekvd@assert@new	.....	80, <u>776</u>
\ekvd@assert@new@for@name	.....	551, 570, 582, 605, 617, 788
\ekvd@assert@not@also	.....	181, 556, 574, 586, 609, 621, 779
\ekvd@assert@not@also@long@also	.....	<u>776</u>
\ekvd@assert@not@long	.....	83, 112, 144, 168, 182, 436, 455, 460, 555, 575, 623, <u>776</u>
\ekvd@assert@not@long@also	.....	437, 456, 475
\ekvd@assert@not@new	143, 167, 180, <u>776</u>	
\ekvd@assert@not@protected	.....	84, 183, 610, 622, <u>776</u>
\ekvd@assert@not@protected@also	.....	94, <u>776</u>
\ekvd@assert@noval	.....	652, <u>716</u>
\ekvd@assert@noval@	.....	<u>716</u>
\ekvd@assert@twoargs	.....	255, 257, 260, 265, 443, <u>706</u>
\ekvd@assert@val	.....	465, 644, <u>716</u>
\ekvd@assert@val@	.....	<u>716</u>
\ekvd@choice@invalid@p	.....	525, 534, 535, 536
\ekvd@choice@invalid@p@	... 527, 530	
\ekvd@choice@name	.....	217, 220, 246, 249, 462, 498, 517, <u>871</u>
\ekvd@choice@p@also	.....	535
\ekvd@choice@p@long	.....	<u>458</u>
\ekvd@choice@p@long@	.....	<u>458</u>
\ekvd@choice@p@new	.....	536
\ekvd@choice@p@protect	.....	<u>458</u>
\ekvd@choice@p@protected	.....	<u>458</u>
\ekvd@choice@prefix	.....	<u>458</u>
\ekvd@choice@prefix@	.....	<u>458</u>
\ekvd@clear@prefixes	.....	27, 53, 491
\ekvd@cur	.....	54, 487, 492, 532, 822, 826, 832, 838, 845, 849, 853, 855, 857, 864, 868, 870
\ekvd@empty	.....	27, 393, 749
\ekvd@err	.....	891, <u>894</u>
\ekvd@err@add@noval@on@val	.. 741, <u>820</u>	
\ekvd@err@add@val@on@noval	.. 722, <u>820</u>	
\ekvd@err@choice@invalid	.. 637, <u>871</u>	
\ekvd@err@choice@invalid@	.....	<u>871</u>
\ekvd@err@missing@definition	...	88, 708, 761, 768, <u>820</u>
\ekvd@err@missing@definition@msg	.....	487, 764, <u>820</u>
\ekvd@err@missing@type	... 57, 74, <u>820</u>	
\ekvd@err@misused@unknown	.. 566, 869	
\ekvd@err@no@prefix	.....	776, 778, 779, 787, <u>820</u>
\ekvd@err@no@prefix@also	781, 783, <u>820</u>	
\ekvd@err@no@prefix@msg	... 532, <u>820</u>	
\ekvd@err@not@new	.....	785, 794, <u>820</u>

<code>\ekvd@err@undefined@key</code> .....	<code>\ekvd@prefix@</code> .....	<u>59</u>
..... 154, 175, 197, 723, 742, <u>820</u>	<code>\ekvd@prefix@after@p</code> .....	<u>66, 70</u>
<code>\ekvd@err@undefined@noval</code> .	<code>\ekvd@prot</code> .....	
..... 188, <u>840</u>	... 27, 77, 113, 128, 145, 169, 277,	
<code>\ekvd@err@undefined@prefix</code> .....	308, 329, 347, 396, 431, 461, 515,	
..... 67, 521, <u>820</u>	523, 557, 576, 592, 666, 681, 778, 783	
<code>\ekvd@err@unsupported@arg</code> .....	<code>\ekvd@set</code> .... 38, 96, 116, 131, 141,	
..... 731, 750, <u>820</u>	148, 152, 165, 170, 173, 186, 187,	
<code>\ekvd@errm</code> .....	191, 194, 215, 217, 220, 244, 246,	
<u>820</u>	249, 281, 312, 332, 351, 371, 399,	
<code>\ekvd@extract@args</code> .....	413, 417, 448, 462, 472, 479, 498,	
<u>716</u>	517, 551, 558, 570, 576, 582, 592,	
<code>\ekvd@extract@prefixes</code> ....	605, 611, 617, 624, 646, 654, 666,	
660, <u>668</u>	718, 719, 721, 737, 738, 740, 785, 800	
<code>\ekvd@extract@prefixes@</code> .....	<code>\ekvd@set@choice</code> .....	<u>498, 517, 544</u>
<u>668</u>	<code>\ekvd@stop</code> . 44, 49, 51, 670, 674, 677,	
<code>\ekvd@extract@prefixes@long</code> ....	680, 683, 685, 729, 748, 756, 801, 807	
<u>668</u>	<code>\ekvd@t@apptoks</code> .....	<u>338</u>
<code>\ekvd@extract@prefixes@prot</code> ....	<code>\ekvd@t@bool</code> .....	<u>205</u>
<u>668</u>	<code>\ekvd@t@boolpair</code> .....	<u>235</u>
<code>\ekvd@extracted@args</code> .....	<code>\ekvd@t@boolpairTF</code> .....	<u>235</u>
<u>716, 864</u>	<code>\ekvd@t@boolTF</code> .....	<u>205</u>
<code>\ekvd@h@choice</code> .... 462, <u>628</u> , 801, 807	<code>\ekvd@t@box</code> .....	<u>299</u>
<code>\ekvd@h@choice@</code> .....	<code>\ekvd@t@choice</code> .....	<u>458</u>
<u>628</u>	<code>\ekvd@t@code</code> .....	<u>122</u>
<code>\ekvd@handle</code> .....	<code>\ekvd@t@data</code> .....	<u>268</u>
<u>41</u>	<code>\ekvd@t@dataT</code> .....	<u>268</u>
<code>\ekvd@if@not@already@choice</code> 467, <u>797</u>	<code>\ekvd@t@default</code> .....	<u>137</u>
<code>\ekvd@if@not@already@choice@a</code> ..	<code>\ekvd@t@dimen</code> .....	<u>359</u>
<u>797</u>	<code>\ekvd@t@ecode</code> .....	<u>122</u>
<code>\ekvd@if@not@already@choice@b</code> ..	<code>\ekvd@t@edata</code> .....	<u>289</u>
<u>797</u>	<code>\ekvd@t@edataT</code> .....	<u>295</u>
<code>\ekvd@ifalso</code> .....	<code>\ekvd@t@edefault</code> .....	<u>161</u>
27,	<code>\ekvd@t@edimen</code> .....	<u>359</u>
79, 90, 114, 129, 150, 171, 275, 306,	<code>\ekvd@t@einitial</code> .....	<u>178</u>
327, 345, 366, 394, 415, 446, 463, 779	<code>\ekvd@t@eint</code> .....	<u>359</u>
<code>\ekvd@ifempty@gtwo</code> .....	<code>\ekvd@t@enoval</code> .....	<u>106</u>
<u>706</u>	<code>\ekvd@t@eskip</code> .....	<u>359</u>
<code>\ekvd@ifnew</code> ... 33, 80, 85, 108, 124,	<code>\ekvd@t@estore</code> .....	<u>405</u>
207, 209, 237, 239, 270, 301, 322,	<code>\ekvd@t@fdefault</code> .....	<u>137</u>
340, 361, 389, 409, 441, 539, 787, 790	<code>\ekvd@t@finitial</code> .....	<u>178</u>
<code>\ekvd@ifnoarg</code> 43, 48, 102, 184, <u>761</u> , 772	<code>\ekvd@t@gapptoks</code> .....	<u>338</u>
<code>\ekvd@ifnoarg@or@empty</code> .....	<code>\ekvd@t@gbool</code> .....	<u>205</u>
<u>766</u>	<code>\ekvd@t@gboolpair</code> .....	<u>235</u>
<code>\ekvd@ifnottwoargs</code> .....	<code>\ekvd@t@gboolpairTF</code> .....	<u>235</u>
<u>706</u>	<code>\ekvd@t@gboolTF</code> .....	<u>205</u>
<code>\ekvd@ifspace</code> .....	<code>\ekvd@t@gbox</code> .....	<u>299</u>
..... 55, 72, 494, 512, 525, 528, <u>811</u>	<code>\ekvd@t@gdata</code> .....	<u>268</u>
<code>\ekvd@ifspace@</code> .....	<code>\ekvd@t@gdataT</code> .....	<u>268</u>
<u>811</u>	<code>\ekvd@t@gdimen</code> .....	<u>359</u>
<code>\ekvd@long</code> .....		
27,		
76, 128, 281, 312, 332, 351, 371,		
399, 431, 475, 592, 647, 678, 776, 781		
<code>\ekvd@mark</code> .....		
677, 680, 683, 685		
<code>\ekvd@newlet</code> 213, 241, 242, 274, 393, <u>698</u>		
<code>\ekvd@newreg</code> ... 305, 326, 344, 365, <u>698</u>		
<code>\ekvd@noarg</code> .....		
39, <u>41</u>		
<code>\ekvd@one@arg@string</code> .....		
<u>716</u>		
<code>\ekvd@p@also</code> .....		
<u>76</u>		
<code>\ekvd@p@long</code> .....		
<u>76</u>		
<code>\ekvd@p@new</code> .....		
<u>76</u>		
<code>\ekvd@p@protect</code> .....		
<u>76</u>		
<code>\ekvd@p@protected</code> .....		
<u>76</u>		
<code>\ekvd@populate@choice</code> .....		
<u>458</u>		
<code>\ekvd@populate@choice@</code> .....		
<u>458</u>		
<code>\ekvd@populate@choice@noarg</code> ....		
<u>458</u>		
<code>\ekvd@prefix</code> .....		
56, <u>59</u> , <u>73</u>		

<code>\ekvd@t@gint</code>	<u>359</u>	<code>\ekvd@type@default</code>	<u>137</u>
<code>\ekvd@t@ginvbool</code>	<u>205</u>	<code>\ekvd@type@initial</code>	.....
<code>\ekvd@t@ginvboolTF</code>	<u>205</u>	.....	178, 200, 201, 202, 204
<code>\ekvd@t@gskip</code>	<u>359</u>	<code>\ekvd@type@meta</code>	<u>407</u>
<code>\ekvd@t@gstore</code>	<u>387</u>	<code>\ekvd@type@meta@a</code>	<u>407</u> , 445
<code>\ekvd@t@gtoks</code>	<u>320</u>	<code>\ekvd@type@meta@b</code>	<u>407</u>
<code>\ekvd@t@initial</code>	<u>178</u>	<code>\ekvd@type@meta@c</code>	<u>407</u>
<code>\ekvd@t@int</code>	<u>359</u>	<code>\ekvd@type@noval</code>	<u>106</u>
<code>\ekvd@t@invbool</code>	<u>205</u>	<code>\ekvd@type@reg</code>	<u>359</u>
<code>\ekvd@t@invboolTF</code>	<u>205</u>	<code>\ekvd@type@set</code>	<u>81</u>
<code>\ekvd@t@meta</code>	<u>407</u>	<code>\ekvd@type@smeta</code>	<u>439</u>
<code>\ekvd@t@nmeta</code>	<u>407</u>	<code>\ekvd@type@smeta@</code>	<u>439</u>
<code>\ekvd@t@noval</code>	<u>106</u>	<code>\ekvd@type@store</code>	<u>387</u>
<code>\ekvd@t@odefault</code>	<u>137</u>	<code>\ekvd@type@toks</code>	<u>320</u>
<code>\ekvd@t@oinitial</code>	<u>178</u>	<code>\ekvd@type@unknown@code</code>	.....
<code>\ekvd@t@qdefault</code>	<u>137</u>	.....	562
<code>\ekvd@t@set</code>	<u>81</u>	<code>\ekvd@type@unknown@noval</code>	.....
<code>\ekvd@t@skip</code>	<u>359</u>	.....	562
<code>\ekvd@t@smeta</code>	<u>439</u>	<code>\ekvd@type@unknown@redirect</code>	....
<code>\ekvd@t@snmeta</code>	<u>439</u>	.....	598
<code>\ekvd@t@store</code>	<u>387</u>	<code>\ekvd@type@unknown@redirect-code</code>	<u>598</u>
<code>\ekvd@t@toks</code>	<u>320</u>	<code>\ekvd@type@unknown@redirect-noval</code>	.....
<code>\ekvd@t@unknown</code>	<u>562</u>	.....	598
<code>\ekvd@t@unknown-choice</code>	<u>549</u>	<code>\ekvd@unknown@choice@name</code>	.....
<code>\ekvd@t@xdata</code>	<u>292</u>	.....	551, 558, 871
<code>\ekvd@t@xdataT</code>	<u>298</u>	<code>\evkd@prot</code>	<u>368</u>
<code>\ekvd@t@xdimen</code>	<u>359</u>	<code>toks</code>	.....
<code>\ekvd@t@xint</code>	<u>359</u>	.....	6
<code>\ekvd@t@xskip</code>	<u>359</u>	<b>U</b>	
<code>\ekvd@t@xstore</code>	<u>406</u>	<code>unknown_ code</code>	.....
<code>\ekvd@tmp</code>	<u>2, 113, 115, 116,</u> <u>128, 130, 131, 145, 151, 152, 169,</u> <u>172, 173, 195, 200, 201, 202, 204,</u> <u>413, 414, 416, 417, 431, 447, 448,</u> <u>461, 474, 479, 588, 594, 689, 697, 895</u>	<code>unknown_ noval</code>	.....
<code>\ekvd@type@apptoks</code>	<u>338</u>	<code>unknown_ redirect</code>	.....
<code>\ekvd@type@bool</code>	<u>205</u>	<code>unknown_ redirect-code</code>	.....
<code>\ekvd@type@boolpair</code>	<u>235</u>	<code>unknown_ redirect-noval</code>	.....
<code>\ekvd@type@box</code>	<u>299</u>	<code>unknown-choice</code>	.....
<code>\ekvd@type@choice</code>	..... 214, 243, <u>458</u>	<code>\unprotect</code>	.....
<code>\ekvd@type@code</code>	<u>122</u>	<code>\usemodule</code>	.....
<code>\ekvd@type@ddata</code>	<u>268</u>	.....	11
		<b>W</b>	
		<code>\writestatus</code>	..... 10, 14
		<b>X</b>	
		<code>xdata</code>	.....
		<code>xdataT</code>	.....
		<code>xdimen</code>	.....
		<code>xint</code>	.....
		<code>xskip</code>	.....
		<code>xstore</code>	.....