

expkv|DEF

a key-defining frontend for **expkv**

Jonathan P. Spratte*

2020-07-12 vo.5

Abstract

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

Contents

1	Documentation	2
1.1	Macros	2
1.2	Prefixes	2
1.2.1	p-Prefixes	2
1.2.2	t-Prefixes	3
1.3	Bugs	6
1.4	Example	6
1.5	License	8
2	Implementation	9
2.1	The L ^A T _E X Package	9
2.2	The Generic Code	9
2.2.1	Key Types	11
2.2.2	Key Type Helpers	21
2.2.3	Handling also	21
2.2.4	Tests	23
2.2.5	Messages	25

Index	28
--------------	-----------

*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., `.tl_set:N` of `l3keys`) or directory like handlers (e.g., `./store` in of `pgfkeys`). This was decided as a personal preference, more over in `TEX` parsing for the first space is way easier than parsing for the last one. `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 [subsubsection 1.2.1](#), the t-prefixes are described in [subsubsection 1.2.2](#).

`expkvdef` is usable as generic code and as a `LATeX` package. It'll automatically load `expkv` in the same mode as well. To use it, just use one of

```
\usepackage{expkv-def} % LaTeX
\input expkv-def       % plainTeX
```

1.1 Macros

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

```
\ekvdefinekeys
```

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 `also` might be simple on first sight as well, but its 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
}
```

**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`.

1.2.2 t-Prefixes

Since the p-type prefixes apply to some of the t-prefixes automatically but sometimes one might be disallowed we need some way to highlight this behaviour. In the following an enforced prefix will be printed black (`protected`), allowed prefixes will be grey (`protected`), and disallowed prefixes will be red (`protected`). This will be put flush-right in the syntax showing line.

**code
ecode**

```
code <key> = {{definition}}                                also protected long
```

Define `<key>` to expand to `<definition>`. The `<key>` will require a `<value>` for which you can use `#1` inside `<definition>`. The `ecode` variant will fully expand `<definition>` inside an `\edef`.

**noval
enoval**

```
noval <key> = {{definition}}                                also protected long
```

The `noval` type defines `<key>` to expand to `<definition>`. The `<key>` will not take a `<value>`. `enoval` fully expands `<definition>` inside an `\edef`.

```
default <key> = {{definition}}                                also protected long
```

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

```
initial <key> = {<value>}                                also protected long
```

With `initial` you can set an initial *value* for an already defined argument taking *key*. It'll just call the key-macro of *key* and pass it *value*. The `einitial` variant will expand *value* using an `\edef` expansion prior to passing it to the key-macro and the `oinitial` variant will expand the first token in *value* once.

```
bool <key> = <cs>                                         also protected long
```

The *cs* should be a single control sequence, such as `\iffoo`. This will define *key* to be a boolean key, which only takes the values `true` or `false` and will throw an error for other values. If the key is used without a *value* it'll have the same effect as if you use *key*=`true`. `bool` and `gbool` will behave like TeX-ifs so either be `\iftrue` or `\iffalse`. The `boolTF` and `gboolTF` variants will both take two arguments and if true the first will be used else the second, so they are always either `\@firstoftwo` or `\@secondoftwo`. The variants with a leading `g` will set the control sequence globally, the others locally. If *cs* is not yet defined it'll be initialised as the `false` version. Note that the initialisation is *not* done with `\newif`, so you will not be able to do `\foottrue` outside of the *key*=*value* interface, but you could use `\newif` yourself. Even if the *key* will not be `\protected` the commands which execute the `true` or `false` choice will be, so the usage should be safe in an expansion context (e.g., you can use `edefault <key> = false` without an issue to change the default behaviour to execute the `false` choice). Internally a `bool` *key* is the same as a choice key which is set up to handle `true` and `false` as choices.

```
store <key> = <cs>                                         also protected long
```

The *cs* should be a single control sequence, such as `\foo`. This will define *key* to store *value* inside of the control sequence. If *cs* isn't yet defined it will be initialised as empty. The variants behave similarly to their `\def`, `\edef`, `\gdef`, and `\xdef` counterparts, but `store` and `gstore` will allow you to store macro parameters inside of them by using `\unexpanded`.

```
data <key> = <cs>                                         also protected long
```

The *cs* should be a single control sequence, such as `\foo`. This will define *key* to store *value* inside of the control sequence. But unlike the `store` type, the macro *cs* will be a switch at the same time, it'll take two arguments and if *key* was used expands to the first argument followed by *value* in braces, if *key* was not used *cs* will expand to the second argument (so behave like `\@secondoftwo`). The idea is that with this type you can define a key which should be typeset formatted. The `edata` and `xdata` variants will fully expand *value*, the `gdata` and `xdata` variants will store *value* inside *cs* globally. The p-prefixes will only affect the key-macro, *cs* will always be expandable and `\long`.

<u>dataT</u>	<code>dataT <key> = <cs></code>	<code>also protected long</code>
<u>edataT</u>	Just like data, but instead of <code><cs></code> grabbing two arguments it'll only grab one, so by default it'll behave like <code>\@gobble</code> , and if a <code><value></code> was given to <code><key></code> the <code><cs></code> will behave like <code>\@firstofone</code> appended by <code>{<value>}</code> .	
<u>gdataT</u>		
<u>xdataT</u>		
<u>int</u>	<code>int <key> = <cs></code>	<code>also protected long</code>
<u>eint</u>	The <code><cs></code> should be a single control sequence, such as <code>\foo</code> . An int key will be a TeX-count register. If <code><cs></code> isn't defined yet, <code>\newcount</code> will be used to initialise it. The eint and xint versions will use <code>\numexpr</code> to allow basic computations in their <code><value></code> . The gint and xint variants set the register globally.	
<u>gint</u>		
<u>xint</u>		
<u>dimen</u>	<code>dimen <key> = <cs></code>	<code>also protected long</code>
<u>edimen</u>	The <code><cs></code> should be a single control sequence, such as <code>\foo</code> . This is just like int but uses a dimen register, <code>\newdimen</code> and <code>\dimexpr</code> instead.	
<u>gdimen</u>		
<u>xdimen</u>		
<u>skip</u>	<code>skip <key> = <cs></code>	<code>also protected long</code>
<u>eskip</u>	The <code><cs></code> should be a single control sequence, such as <code>\foo</code> . This is just like int but uses a skip register, <code>\newskip</code> and <code>\glueexpr</code> instead.	
<u>gskip</u>		
<u>xskip</u>		
<u>toks</u>	<code>toks <key> = <cs></code>	<code>also protected long</code>
<u>gtoks</u>	The <code><cs></code> should be a single control sequence, such as <code>\foo</code> . Store <code><value></code> inside of a toks-register. The g variants use <code>\global</code> , the app variants append <code><value></code> to the contents of that register. If <code><cs></code> is not yet defined it will be initialised with <code>\newtoks</code> .	
<u>apptoks</u>		
<u>gapptoks</u>		
<u>box</u>	<code>box <key> = <cs></code>	<code>also protected long</code>
<u>gbox</u>	The <code><cs></code> should be a single control sequence, such as <code>\foo</code> . Typesets <code><value></code> into a <code>\hbox</code> and stores the result in a box register. The boxes are colour safe. <code>\expKV\DEF</code> doesn't provide a <code>vbox</code> type.	
<u>meta</u>	<code>meta <key> = {<key>=<value>, ...}</code>	<code>also protected long</code>
	This key type can set other keys, you can access the <code><value></code> which was passed to <code><key></code> inside the <code><key>=<value></code> list with #1. It works by calling a sub- <code>\ekvset</code> on the <code><key>=<value></code> list, so a <code>set</code> key will only affect that <code><key>=<value></code> list and not the current <code>\ekvset</code> . Since it runs in a separate <code>\ekvset</code> you can't use <code>\ekvsneak</code> using keys or similar macros in the way you normally could.	
<u>nmeta</u>	<code>nmeta <key> = {<key>=<value>, ...}</code>	<code>also protected long</code>
	This key type can set other keys, the difference to meta is, that this key doesn't take a value, so the <code><key>=<value></code> list is static.	
<u>smeta</u>	<code>smeta <key> = {<set>}{{<key>=<value>, ...}}</code>	<code>also protected long</code>
	Yet another meta variant. An smeta key will take a <code><value></code> which you can access using #1, but it sets the <code><key>=<value></code> list inside of <code><set></code> , so is equal to <code>\ekvset{<set>}{{<key>=<value>, ...}}</code> .	

`snmeta` `<key> = {{<set>}}{<key>}=<value>, ...` also protected long

And the last meta variant. `snmeta` is a combination of `smeta` and `nmeta`. It doesn't take an argument and sets the `<key>=<value>` list inside of `<set>`.

`set` `<key> = {{<set>}}` 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` `<key> = {{<value>}}=<definition>, ...` 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` `<key> = {{<definition>}}` also protected long

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

1.3 Bugs

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

1.4 Example

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

```
\ekvdefinekeys{example}
{
    long code keyA = #1
    ,noval      keyA = NoVal given
    ,bool       keyB = \keyB
    ,boolTF     keyC = \keyC
    ,store      keyD = \keyD
    ,data       keyE = \keyE
    ,dataT     keyF = \keyF
    ,int        keyG = \keyG
```

```

, dimen      keyH = \keyH
, skip       keyI = \keyI
, toks       keyJ = \keyJ
, default    keyJ = \empty test
, box        keyK = \keyK
, qdefault   keyK = text
, choice     keyL =
{
  protected 1 = \texttt{a}
  ,2 = b
  ,3 = c
  ,4 = d
  ,5 = e
}
, edefault   keyL = 2
, meta       keyM = {keyA={#1},keyB=false}
}

```

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

```

\ekvdefinekeys{ex}
{
  data name = \Pname
  ,data age = \Page
  ,dataT hobby = \Phobby
}
\newcommand\Person[1]
{%
  \begingroup
  \ekvset{ex}{#1}%
  \begin{description}
    \item[\Pname{}]\errmessage{A person requires a name{}}
    \item[Age] \Page{\textit{}}\errmessage{A person requires an age{}}
    \Phobby{\item[Hobbies]}
  \end{description}
  \endgroup
}
\Person{name=Jonathan P. Spratte, age=young, hobby=\TeX\ coding}
\Person{name=Some User, age=unknown, hobby=Reading Documentation}
\Person{name=Anybody, age=any}

```

In this example a person should have a name and an age, but doesn't have to have hobbies. The name will be displayed as the description item and the age in Italics. If a person has no hobbies the description item will be silently left out. The result of the above code looks like this:

Jonathan P. Spratte

Age *young*

Hobbies \TeX coding

Some User

Age *unknown*

Hobbies Reading Documentation

Anybody

Age *any*

1.5 License

Copyright © 2020 Jonathan P. Spratte

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

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

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

2 Implementation

2.1 The L^AT_EX Package

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

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

2.2 The Generic Code

The rest of this implementation will be the generic code.

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

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

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

```
\ekvdDate
15 \def\ekvdVersion{0.5}
16 \def\ekvdDate{2020-07-12}
```

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

If the L^AT_EX format is loaded we want to be a good file and report back who we are, for this the package will have defined `\ekvd@tmp` to use `\ProvidesFile`, else this will expand to a `\relax` and do no harm.

```
17 \csname ekvd@tmp\endcsname
    Store the category code of @ to later be able to reset it and change it to 11 for now.
18 \expandafter\chardef\csname ekvd@tmp\endcsname=\catcode`\@
19 \catcode`\@=11
```

`\ekvd@tmp` will be reused later to handle expansion during the key defining. But we don't need it to ever store information long-term after `expkv|DEF` was initialized.

```
\ekvd@long
\ekvd@prot
\ekvd@clear@prefixes
\ekvd@empty
\ekvd@ifalso
20 \def\ekvd@empty{}  

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

```

21 \protected\def\ekvd@clear@prefixes
22 {%
23   \let\ekvd@long\ekvd@empty
24   \let\ekvd@prot\ekvd@empty
25   \let\ekvd@ifalso@\secondoftwo
26 }
27 \ekvd@clear@prefixes

```

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

`\ekvd@exp@Nno` These are expansion helpers, similar to what L^AT_EX 3 uses, but simpler and for just a few cases

```

28 \long\def\ekvd@exp@Nno#1#2#3%
29 {%
30   \expandafter\ekvd@exp@reinsert@n\expandafter{#3}{#1{#2}}%
31 }
32 \long\def\ekvd@exp@reinsert@n#1#2{#2{#1}}

```

(End definition for `\ekvd@exp@Nno` and `\ekvd@exp@reinsert@n`.)

\ekvdefinekeys This is the one front-facing macro which provides the interface to define keys. It's using `\ekvpars`e 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`.

```

33 \protected\def\ekvdefinekeys#1%
34 {%
35   \def\ekvd@set{#1}%
36   \ekvpars\ekvd@noarg\ekvd@
37 }

```

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

`\ekvd@noarg` `\ekvd@` `\ekvd@noarg` just places a special marker and gives control to `\ekvd@`. `\ekvd@` 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.

```

38 \protected\def\ekvd@noarg#1{\ekvd@{#1}\ekvd@noarg@mark}
39 \protected\long\def\ekvd@#1#2%
40 {%
41   \ekvd@clear@prefixes
42   \edef\ekvd@cur{\detokenize{#1}}%
43   \ekvd@ifspace{#1}%
44   {\ekvd@prefix\ekv@mark#1\ekv@stop{#2}}%
45   \ekvd@err@missing@type
46 }

```

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

`\ekvd@prefix` `\ekvd@prefix@` **expKV|DEF** separates prefixes into two groups, the first being prefixes in the T_EX 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).

```

47 \protected\def\ekvd@prefix#1 {\ekv@strip{#1}\ekvd@prefix@\ekv@mark}
48 \protected\def\ekvd@prefix@#1#2\ekv@stop

```

```

49   {%
50     \ekv@ifdefined{ekvd@t@#1}%
51       {\ekv@strip{#2}{\csname ekvd@t@#1\endcsname}}%
52     {%
53       \ekv@ifdefined{ekvd@p@#1}%
54         {\csname ekvd@p@#1\endcsname\ekvd@prefix@after@p{#2}}%
55         {\ekvd@err@undefined@prefix{#1}\@gobble}%
56     }%
57   }

```

(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.

```

58 \protected\def\ekvd@prefix@after@p{#1}%
59 {%
60   \ekvd@ifspace{#1}%
61   {\ekvd@prefix{#1}\ekv@stop}%
62   {\ekvd@err@missing@type\@gobble}%
63 }

```

(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.

```

64 \protected\def\ekvd@p@long{\let\ekvd@long\long}
65 \protected\def\ekvd@p@protected{\let\ekvd@prot\protected}
66 \let\ekvd@p@protect\ekvd@p@protected
67 \protected\def\ekvd@p@also{\let\ekvd@ifalso\@firstoftwo}

```

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

2.2.1 Key Types

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

```

68 \protected\def\ekvd@t@set{#1}%
69 {%
70   \ekvd@assert@not@long
71   \ekvd@assert@not@protected
72   \ekv@ifempty{#2}%
73     {\ekvd@err@missing@definition}%
74   {%
75     \ekvd@ifalso
76     {%
77       \ekvd@ifnoarg{#2}%
78       {\ekvd@add@noval{#1}{\ekvchangeset{#1}}}%
79       {\ekvd@add@noval{#1}{\ekvchangeset{#2}}}%
80       \ekvd@assert@not@protected@also
81     }%
82   {%
83     \ekvd@ifnoarg{#2}%
84     {\ekvdefNoVal\ekvd@set{#1}{\ekvchangeset{#1}}}%
85     {\ekvdefNoVal\ekvd@set{#1}{\ekvchangeset{#2}}}%
86   }%

```

```

87     }%
88 }
```

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

`\ekvd@type@noval` 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.

```

89 \protected\long\def\ekvd@type@noval#1#2#3#4%
90   {%
91     \ekvd@assert@arg{#4}%
92     {%
93       \ekvd@assert@not@long
94       \ekvd@prot#2\ekvd@tmp{#4}%
95       \ekvd@ifalso
96         {\ekvd@exp@Nno\ekvd@add@noval{#3}\ekvd@tmp{}{}}%
97         {\ekvletNoVal\ekvd@set{#3}\ekvd@tmp{}{}}%
98     }%
99   }
100 \protected\def\ekvd@t@noval{\ekvd@type@noval{}\def}
101 \protected\def\ekvd@t@enoval{\ekvd@type@noval e\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 provide an `\ekvedef`.

```

102 \protected\long\def\ekvd@type@code#1#2#3#4%
103   {%
104     \ekvd@assert@arg{#4}%
105     {%
106       \ekvd@prot\ekvd@long#2\ekvd@tmp##1{#4}%
107       \ekvd@ifalso
108         {\ekvd@exp@Nno\ekvd@add@val{#3}{\ekvd@tmp{##1}}{}{}}%
109         {\ekvlet\ekvd@set{#3}\ekvd@tmp{}{}}%
110     }%
111   }
112 \protected\def\ekvd@t@code{\ekvd@type@code{}\def}
113 \protected\def\ekvd@t@ecode{\ekvd@type@code e\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.

```

114 \protected\long\def\ekvd@type@default#1#2#3#4%
115   {%
116     \ekvd@assert@arg{#4}%
117     {%
118       \ekvifdefined\ekvd@set{#3}%
119         {%
```

```

120   \ekvd@assert@not@long
121   \ekvd@prot\edef\ekvd@tmp
122   {%
123     \unexpanded\expandafter#2%
124     {\csname\ekv@name\ekvd@set{\#3}\endcsname{\#4}}%
125   }%
126   \ekvd@ifalso
127     {\ekvd@exp@Nno\ekvd@add@noval{\#3}\ekvd@tmp{}%}
128     {\ekvletNoVal\ekvd@set{\#3}\ekvd@tmp}%
129   }%
130   {\ekvd@err@undefined@key{\#3}}%
131 }%
132 }
133 \protected\def\ekvd@t@default{\ekvd@type@default{}{}}
134 \protected\def\ekvd@t@qdefault{\ekvd@type@default q{\expandafter\expandafter}}
```

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

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

```

135 \protected\long\def\ekvd@t@edefault#1#2%
136 {%
137   \ekvd@assert@arg{\#2}%
138   {%
139     \ekvifdefined\ekvd@set{\#1}%
140     {%
141       \ekvd@assert@not@long
142       \ekvd@prot\edef\ekvd@tmp
143         {\csname\ekv@name\ekvd@set{\#1}\endcsname{\#2}}%
144     \ekvd@ifalso
145       {\ekvd@exp@Nno\ekvd@add@noval{\#1}\ekvd@tmp{}%}
146       {\ekvletNoVal\ekvd@set{\#1}\ekvd@tmp}%
147     }%
148     {\ekvd@err@undefined@key{\#1}}%
149   }%
150 }
```

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

```

\ekvd@t@initial
\ekvd@t@oinitial
\ekvd@t@einitial
151 \long\def\ekvd@t@initial#1#2%
152 {%
153   \ekvd@assert@arg{\#2}%
154   {%
155     \ekvifdefined\ekvd@set{\#1}%
156     {%
157       \ekvd@assert@not@also
158       \ekvd@assert@not@long
159       \ekvd@assert@not@protected
160       \csname\ekv@name\ekvd@set{\#1}\endcsname{\#2}}%
161     }%
162     {\ekvd@err@undefined@key{\#1}}%
163   }%
164 }
```

```

165 \long\def\ekvd@t@oinitial#1#2%
166   {%
167     \ekvd@assert@arg{#2}%
168   {%
169     \ekvifdefined\ekvd@set{#1}%
170   {%
171     \ekvd@assert@not@also
172     \ekvd@assert@not@long
173     \ekvd@assert@not@protected
174     \csname\ekv@name\ekvd@set{#1}\expandafter\endcsname\expandafter{#2}%
175   }%
176   {\ekvd@err@undefined@key{#1}}%
177 }%
178 }
179 \long\def\ekvd@t@einitial#1#2%
180   {%
181     \ekvd@assert@arg{#2}%
182   {%
183     \ekvifdefined\ekvd@set{#1}%
184   {%
185     \ekvd@assert@not@also
186     \ekvd@assert@not@long
187     \ekvd@assert@not@protected
188     \edef\ekvd@tmp{#2}%
189     \csname\ekv@name\ekvd@set{#1}\expandafter\endcsname\expandafter
190       {\ekvd@tmp}%
191   }%
192   {\ekvd@err@undefined@key{#1}}%
193 }%
194 }

```

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

```

\ekvd@type@bool
\ekvd@t@bool
\ekvd@t@gbool
\ekvd@t@boolTF
\ekvd@t@gboolTF

```

The boolean types are a quicker version of a choice that accept `true` and `false`, and set up the `NoVal` action to be identical to `<key>=true`. The `true` and `false` actions are always just `\letting` the macro in #7 to some other macro (e.g., `\iftrue`).

```

195 \protected\def\ekvd@type@bool#1#2#3#4#5#6#7%
196   {%
197     \ekvd@assert@filledarg{#7}%
198   {%
199     \ekvd@newlet#7#5%
200     \ekvd@type@choice{#1bool#2}{#6}%
201     \protected\ekvdefNoVal\ekvd@set{#6}{#3\let#7#4}%
202     \protected\expandafter\def
203       \csname\ekvd@choice@name\ekvd@set{#6}{true}\endcsname
204       {#3\let#7#4}%
205     \protected\expandafter\def
206       \csname\ekvd@choice@name\ekvd@set{#6}{false}\endcsname
207       {#3\let#7#5}%
208   }%
209 }
210 \protected\def\ekvd@t@bool{\ekvd@type@bool{}{}{}\iftrue\iffalse}
211 \protected\def\ekvd@t@gbool{\ekvd@type@bool g{}\global\iftrue\iffalse}
212 \protected\def\ekvd@t@boolTF{\ekvd@type@bool{}{TF}{}\@firstoftwo\@secondoftwo}

```

```

213 \protected\def\ekvd@t@gboolTF
214   {\ekvd@type@bool g{TF}\global\@firstoftwo\@secondoftwo}

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

\ekvd@type@data
215 \protected\def\ekvd@type@data#1#2#3#4#5#6#7#8%
216   {%
217     \ekvd@assert@filledarg{#8}%
218     {%
219       \ekvd@newlet#8#3%
220       \ekvd@ifalso
221       {%
222         \let\ekvd@prot\protected
223         \ekvd@add@val{#7}{\long#4#8####1#5{####1{#6}}}{}
224       }%
225       {%
226         \protected\ekvd@long\ekvdef\ekvd@set{#7}%
227         {\long#4#8####1#5{####1{#6}}}{}
228       }%
229     }%
230   }%
231 \protected\def\ekvd@t@data
232   {\ekvd@type@data{}@\secondoftwo\edef{####2}{\unexpanded{##1}}{}}
233 \protected\def\ekvd@t@edata{\ekvd@type@data e{}@\secondoftwo\edef{####2}{##1}}
234 \protected\def\ekvd@t@gdata
235   {\ekvd@type@data g{}@\secondoftwo\xdef{####2}{\unexpanded{##1}}{}}
236 \protected\def\ekvd@t@xdata{\ekvd@type@data x{}@\secondoftwo\xdef{####2}{##1}}
237 \protected\def\ekvd@t@dataT{\ekvd@type@data{}T@\gobble\edef{}{\unexpanded{##1}}{}}
238 \protected\def\ekvd@t@edataT{\ekvd@type@data eT@\gobble\edef{}{##1}}%
239 \protected\def\ekvd@t@gdataT
240   {\ekvd@type@data gT@\gobble\xdef{}{\unexpanded{##1}}{}}
241 \protected\def\ekvd@t@xdataTf{\ekvd@type@data xT@\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.

```

242 \protected\def\ekvd@type@box#1#2#3#4%
243   {%
244     \ekvd@assert@filledarg{#4}%
245     {%
246       \ekvd@newreg#4{box}%
247       \ekvd@ifalso
248       {%
249         \let\ekvd@prot\protected
250         \ekvd@add@val{#3}{#2\setbox#4\hbox{\begingroup##1\endgroup}}{}%
251       }%
252       {%
253         \protected\ekvd@long\ekvdef\ekvd@set{#3}%
254         {#2\setbox#4\hbox{\begingroup##1\endgroup}}{}%
255       }%

```

	<pre> 256 }% 257 } 258 \protected\def\ekvd@t@box{\ekvd@type@box{}{}} 259 \protected\def\ekvd@t@gbox{\ekvd@type@box g\global} (End definition for \ekvd@type@box, \ekvd@t@box, and \ekvd@t@gbox.) </pre>
\ekvd@type@toks	Similar to box, but set the toks.
	<pre> 260 \protected\def\ekvd@type@toks#1#2#3#4% 261 {% 262 \ekvd@assert@filledarg{#4}% 263 {% 264 \ekvd@newreg#4{toks}% 265 \ekvd@ifalso 266 {% 267 \let\ekvd@prot\protected 268 \ekvd@add@val{#3}{#2#4{##1}}{}% 269 }% 270 {\protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{##1}}}% 271 }% 272 } 273 \protected\def\ekvd@t@toks{\ekvd@type@toks{}{}} 274 \protected\def\ekvd@t@gtoks{\ekvd@type@toks{g}\global} (End definition for \ekvd@type@toks, \ekvd@t@toks, and \ekvd@t@gtoks.) </pre>
\ekvd@type@apptoks	Just like toks, but expand the current contents of the toks register to append the new contents.
	<pre> 275 \protected\def\ekvd@type@apptoks#1#2#3#4% 276 {% 277 \ekvd@assert@filledarg{#4}% 278 {% 279 \ekvd@newreg#4{toks}% 280 \ekvd@ifalso 281 {% 282 \let\ekvd@prot\protected 283 \ekvd@add@val{#3}{#2#4\expandafter{\the#4##1}}{}% 284 }% 285 {% 286 \protected\ekvd@long\ekvdef\ekvd@set{#3}% 287 {#2#4\expandafter{\the#4##1}}% 288 }% 289 }% 290 } 291 \protected\def\ekvd@t@apptoks{\ekvd@type@apptoks{}{}} 292 \protected\def\ekvd@t@gapptoks{\ekvd@type@apptoks{g}\global} (End definition for \ekvd@type@apptoks, \ekvd@t@apptoks, and \ekvd@t@gapptoks.) </pre>
\ekvd@type@reg	The \ekvd@type@reg can handle all the types for which the assignment will just be <code><register>=(value)</code> .
	<pre> 293 \protected\def\ekvd@type@reg#1#2#3#4#5#6#7% 294 {% 295 \ekvd@assert@filledarg{#7}% 296 {% </pre>

```

297     \ekvd@newreg#7{#2}%
298     \ekvd@ifalso
299     {%
300         \let\evkd@prot\protected
301         \ekvd@add@val{#6}{#3#7=#4##1#5\relax}{}%
302     }%
303     {\protected\ekvd@long\ekvdef\ekvd@set{#6}{#3#7=#4##1#5\relax}}%
304 }%
305 }
306 \protected\def\ekvd@t@int{\ekvd@type@reg{int}{count}{}{}}
307 \protected\def\ekvd@t@eint{\ekvd@type@reg{eint}{count}{}\numexpr\relax}
308 \protected\def\ekvd@t@gint{\ekvd@type@reg{gint}{count}\global{}{}}
309 \protected\def\ekvd@t@xint{\ekvd@type@reg{xint}{count}\global\numexpr\relax}
310 \protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{dimen}{}{}}
311 \protected\def\ekvd@t@edimen{\ekvd@type@reg{edimen}{dimen}{}\dimexpr\relax}
312 \protected\def\ekvd@t@gdimen{\ekvd@type@reg{gdimen}{dimen}\global{}{}}
313 \protected\def\ekvd@t@xdimen{\ekvd@type@reg{xdimen}{dimen}\global\dimexpr\relax}
314 \protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{skip}{}{}}
315 \protected\def\ekvd@t@eskip{\ekvd@type@reg{eskip}{skip}{}\glueexpr\relax}
316 \protected\def\ekvd@t@gskip{\ekvd@type@reg{gskip}{skip}\global{}{}}
317 \protected\def\ekvd@t@xskip{\ekvd@type@reg{xskip}{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.

```

318 \protected\def\ekvd@type@store#1#2#3#4%
319 {%
320     \ekvd@assert@filledarg{#4}%
321     {%
322         \ekvd@newlet#4\ekvd@empty
323         \ekvd@ifalso
324         {%
325             \let\ekvd@prot\protected
326             \ekvd@add@val{#3}{#2#4{\unexpanded{##1}}}{}%
327         }%
328         {\protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{\unexpanded{##1}}}}%
329     }%
330 }
331 \protected\def\ekvd@t@store{\ekvd@type@store{}\edef}
332 \protected\def\ekvd@t@gstore{\ekvd@type@store{g}\xdef}

```

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

`\ekvd@type@estore` And the straight forward `estore` types.

```

333 \protected\def\ekvd@type@estore#1#2#3#4%
334 {%
335     \ekvd@assert@filledarg{#4}%
336     {%
337         \ekvd@newlet#4\ekvd@empty
338         \ekvd@ifalso
339         {%
340             \let\ekvd@prot\protected
341             \ekvd@add@val{#3}{#2#4{##1}}{}%

```

```

342      }%
343      {\protected\ekvd@long\ekvdef\ekvd@set{\#3}{\#2\#4{\##1}}}}%
344      }%
345  }
346 \protected\def\ekvd@t@estore{\ekvd@type@estore{e}\edef}%
347 \protected\def\ekvd@t@xstore{\ekvd@type@estore{x}\xdef}

```

(End definition for `\ekvd@type@estore`, `\ekvd@t@estore`, and `\ekvd@t@xstore`.)

`\ekvd@type@meta` sets up things such that another instance of `\ekvset` will be run on the argument, with the same `<set>`.

```

348 \protected\long\def\ekvd@type@meta{\#1\#2\#3\#4\#5\#6\#7\#8}%
349 {%
350     \ekvd@assert@filledarg{\#8}%
351     {%
352         \edef\ekvd@tmp{\ekvd@set}%
353         \expandafter\ekvd@type@meta@a\expandafter{\ekvd@tmp}{\#8}{\#3}%
354         \ekvd@ifalso
355             {\ekvd@exp@Nno{\#4}{\#7}{\ekvd@tmp{\#5}{\#6}}%
356             {\#2\ekvd@set{\#7}\ekvd@tmp}%
357             }%
358     }%
359 \protected\long\def\ekvd@type@meta@a{\#1\#2}%
360 {%
361     \expandafter\ekvd@type@meta@b\expandafter{\ekvset{\#1}{\#2}}%
362   }%
363 \protected\def\ekvd@type@meta@b
364 {%
365     \expandafter\ekvd@type@meta@c\expandafter
366   }%
367 \protected\long\def\ekvd@type@meta@c{\#1\#2}%
368 {%
369     \ekvd@prot\ekvd@long\def\ekvd@tmp{\#1}%
370   }%
371 \protected\def\ekvd@t@meta{\ekvd@type@meta{}{\ekvlet{\##1}\ekvd@add@val{\{\##1\}}{}}}
372 \protected\def\ekvd@t@nmeta
373 {%
374     \ekvd@assert@not@long
375     \ekvd@type@meta
376     n\ekvletNoVal{}\ekvd@add@noval{}\ekvd@assert@not@long@also
377   }

```

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

`\ekvd@type@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.

```

378 \protected\long\def\ekvd@type@smeta{\#1\#2\#3\#4\#5\#6\#7\#8}%
379 {%
380     \ekvd@assert@twoargs{\#8}%
381     {%
382         \ekvd@type@meta@a{\#8}{\#3}%
383         \ekvd@ifalso
384             {\ekvd@exp@Nno{\#4}{\#7}{\ekvd@tmp{\#5}{\#6}}%
385             {\#2\ekvd@set{\#7}\ekvd@tmp}%

```

```

386      }%
387  }
388 \protected\def\ekvd@t@smeta
389   {\ekvd@type@smeta{}\ekvlet{##1}\ekvd@add@val{{##1}}{}}
390 \protected\def\ekvd@t@snmeta
391   {%
392     \ekvd@assert@not@long
393     \ekvd@type@smeta
394       n\ekvletNoVal{}\ekvd@add@noval{}\ekvd@assert@not@long@also
395   }

```

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

`\ekvd@type@choice`
`\ekvd@populate@choice`
`\ekvd@populate@choice@`
`\ekvd@populate@choice@noarg`
`\ekvd@choice@prefix`
`\ekvd@choice@prefix@`
`\ekvd@choice@p@protected`
`\ekvd@choice@p@protect`
`\ekvd@choice@p@long`
`\ekvd@choice@p@long@`
`\ekvd@t@choice`

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

```

396 \protected\def\ekvd@type@choice#1#2%
397   {%
398     \ekvd@assert@not@long
399     \ekvd@prot\edef\ekvd@tmp##1%
400     {%
401       \unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#2}{##1}}%
402     }%
403     \ekvd@ifalso
404     {%
405       \ekvd@exp@Nno\ekvd@add@val{#2}{\ekvd@tmp{##1}}\ekvd@assert@not@long@also
406     }%
407     {\ekvlet\ekvd@set{#2}\ekvd@tmp}%
408   }

```

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

```

409 \protected\def\ekvd@populate@choice
410   {%
411     \ekvpars\ekvd@populate@choice@noarg\ekvd@populate@choice@
412   }
413 \protected\long\def\ekvd@populate@choice@noarg#1%
414   {%
415     \expandafter\ekvd@err@missing@definition@msg\expandafter{\ekvd@cur : #1}%
416   }

```

`\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.

```

417 \protected\long\def\ekvd@populate@choice@#1#2%
418   {%
419     \ekvd@clear@prefixes
420     \expandafter\ekvd@assert@arg@msg\expandafter{\ekvd@cur : #1}{#2}%
421     {%
422       \ekvd@ifspace{#1}%
423         {\ekvd@choice@prefix\ekv@mark#1\ekv@stop}%

```

```

424     {%
425         \expandafter\def
426             \csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname
427     }%
428     {#2}%
429     }%
430 }
431 \protected\def\ekvd@choice@prefix#1
432 {%
433     \ekv@strip{#1}\ekvd@choice@prefix@{\ekv@mark
434 }
435 \protected\def\ekvd@choice@prefix@#1#2\ekv@stop
436 {%
437     \ekv@ifdefined{\ekvd@choice@p@#1}%
438     {%
439         \csname ekvd@choice@p@#1\endcsname
440         \ekvd@ifspace{#2}%
441         {\ekvd@choice@prefix#2\ekv@stop}%
442     }%
443     \ekvd@prot\expandafter\def
444         \csname
445             \ekv@strip{#2}{\ekvd@choice@name\ekvd@set\ekvd@set@choice}%
446         \endcsname
447     }%
448 }
449 {\ekvd@err@undefined@prefix{#1}\@gobble}%
450 }
451 \protected\def\ekvd@choice@p@protected{\let\ekvd@prot\protected}
452 \let\ekvd@choice@p@protect\ekvd@choice@p@protected
453 \protected\def\ekvd@choice@invalid@p#1\ekvd@ifspace#2%
454 {%
455     \expandafter\ekvd@choice@invalid@p@\expandafter{\ekv@gobble@mark#2}{#1}%
456     \ekvd@ifspace{#2}%
457 }
458 \protected\def\ekvd@choice@invalid@p@#1#2%
459 {%
460     \expandafter\ekvd@err@no@prefix@msg\expandafter{\ekvd@cur : #2 #1}{#2}%
461 }
462 \protected\def\ekvd@choice@p@long{\ekvd@choice@invalid@p{long}}%
463 \protected\def\ekvd@choice@p@also{\ekvd@choice@invalid@p{also}}%

```

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.

```

464 \protected\long\def\ekvd@t@choice#1#2%
465 {%
466     \ekvd@assert@arg{#2}%
467     {%
468         \ekvd@type@choice{choice}{#1}%
469         \def\ekvd@set@choice{#1}%
470         \ekvd@populate@choice{#2}%
471     }%
472 }

```

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

```

\ekvd@t@unknown-choice
473 \protected\long\expandafter\def\csname ekvd@t@unknown-choice\endcsname#1#2%
474 {%
475   \ekvd@assert@arg{#2}%
476   {%
477     \ekvd@assert@not@long
478     \ekvd@assert@not@also
479     \ekvd@prot\expandafter
480     \def\csname\ekvd@unknown@choice@name\ekvd@set{#1}\endcsname##1{#2}%
481   }%
482 }

```

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

2.2.2 Key Type Helpers

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

`\ekvd@h@choice` 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.

```

483 \def\ekvd@h@choice#1%
484 {%
485   \expandafter\ekvd@h@choice@
486   \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname
487   {#1}%
488 }
489 \def\ekvd@h@choice@#1#2%
490 {%
491   \ifx#1\relax
492     \ekvd@err@choice@invalid{#2}%
493     \expandafter\@gobble
494   \fi
495   #1%
496 }

```

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

2.2.3 Handling also

```

\ekvd@add@val
\ekvd@add@noval
\ekvd@add@aux
\ekvd@add@aux@
497 \protected\long\def\ekvd@add@val#1#2#3%
498 {%
499   \ekvd@assert@val{#1}%
500   {%
501     \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}\endcsname{##1}%
502     {#1}{#2}{\ekvd@long\ekvdef}{#3}%
503   }%
504 }
505 \protected\long\def\ekvd@add@noval#1#2#3%
506 {%
507   \ekvd@assert@noval{#1}%
508   {%
509     \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}N\endcsname{}%

```

```

510      {#1}{#2}\ekvd@defNoVal{#3}%
511      }%
512  }
513 \protected\long\def\ekvd@add@aux#1#2%
514  {%
515      \ekvd@extract@prefixes#1%
516      \expandafter\ekvd@add@aux@\expandafter{#1#2}%
517  }
518 \protected\long\def\ekvd@add@aux@#1#2#3#4#5%
519  {%
520      #5%
521      \ekvd@prot#4\ekvd@set{#2}{#1#3}%
522  }

```

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

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

```

523 \protected\def\ekvd@extract@prefixes#1%
524  {%
525      \expandafter\ekvd@extract@prefixes@\meaning#1\ekvd@stop
526  }

```

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.

```

527 \protected\def\ekvd@extract@prefixes@#1#2#3%
528  {%
529      \protected\def\ekvd@extract@prefixes@##1#1##2\ekvd@stop
530  {%
531      \ekvd@extract@prefixes@long
532      ##1\ekvd@mark@firstofone#2\ekvd@mark@gobble\ekvd@stop
533      {\let\ekvd@long\long}%
534      \ekvd@extract@prefixes@prot
535      ##1\ekvd@mark@firstofone#3\ekvd@mark@gobble\ekvd@stop
536      {\let\ekvd@prot\protected}%
537  }%
538 \protected\def\ekvd@extract@prefixes@long##1#2##2\ekvd@mark##3##4\ekvd@stop
539  {##3}%
540 \protected\def\ekvd@extract@prefixes@prot##1#3##2\ekvd@mark##3##4\ekvd@stop
541  {##3}%
542 }

```

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.

```

543 \begingroup
544 \edef\ekvd@tmp
545  {%
546      \endgroup
547      \ekvd@extract@prefixes@
548      {\detokenize{macro:}}%
549      {\string\long}%
550      {\string\protected}%
551  }
552 \ekvd@tmp

```

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

2.2.4 Tests

\ekvd@newlet
\ekvd@newreg

These macros test whether a control sequence is defined, if it isn't they define it, either via \let or via the correct \new.

```
553 \protected\def\ekvd@newlet#1#2%
554   {%
555     \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\let#1#2}%
556   }
557 \protected\def\ekvd@newreg#1#2%
558   {%
559     \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\csname new#2\endcsname#1}%
560   }
```

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

\ekvd@assert@twoargs
\ekvd@ifnottwoargs
\ekvd@ifempty@gtwo

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

```
561 \long\def\ekvd@assert@twoargs#1%
562   {%
563     \ekvd@ifnottwoargs{#1}{\ekvd@err@missing@definition}%
564   }
565 \long\def\ekvd@ifnottwoargs#1%
566   {%
567     \ekvd@ifempty@gtwo#1\ekv@ifempty@B
568       \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
569   }
570 \long\def\ekvd@ifempty@gtwo#1#2{\ekv@ifempty@\ekv@ifempty@A}
```

(End definition for \ekvd@assert@twoargs, \ekvd@ifnottwoargs, and \ekvd@ifempty@gtwo.)

\ekvd@assert@val
\ekvd@assert@val@

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

```
571 \protected\def\ekvd@assert@val#1%
572   {%
573     \ekvifdefined\ekvd@set{#1}%
574       {\expandafter\ekvd@assert@val@\csname\ekv@name\ekvd@set{#1}\endcsname}%
575     {%
576       \ekvifdefinedNoVal\ekvd@set{#1}%
577         \ekvd@err@add@val@on@noval
578           {\ekvd@err@undefined@key{#1}}%
579         \@gobble
580     }%
581   }
582 \protected\def\ekvd@assert@val@#1%
583   {%
584     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
585     \unless\ifx\ekvd@extracted@args\ekvd@one@arg@string
586       \ekvd@err@unsupported@arg
587     \fi
588     \@firstofone
```

```

589   }%
590 \protected\def\ekvd@assert@noval#1%
591   {%
592     \ekvifdefinedNoVal\ekvd@set{#1}%
593       {\expandafter\ekvd@assert@noval@\csname\ekv@name\ekvd@set{#1}N\endcsname}%
594       {%
595         \ekvifdefined\ekvd@set{#1}%
596           \ekvd@err@add@noval@on@val
597           {\ekvd@err@undefined@key{#1}}%
598           \gobble
599       }%
600   }
601 \protected\def\ekvd@assert@noval@#1%
602   {%
603     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
604     \unless\ifx\ekvd@extracted@args\ekvd@empty
605       \ekvd@err@unsupported@arg
606     \fi
607     \firstofone
608   }
609 \protected\def\ekvd@extract@args#1%
610   {%
611     \protected\def\ekvd@extract@args##1##2->##3\ekvd@stop
612       {\def\ekvd@extracted@args{##2}}%
613   }
614 \expandafter\ekvd@extract@args\expandafter{\detokenize{macro:}}
615 \edef\ekvd@one@arg@string{\string#1}

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

```

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

```

616 \long\def\ekvd@assert@arg#1{\ekvd@ifnoarg{#1}\ekvd@err@missing@definition}
617 \long\def\ekvd@assert@arg@msg#1#2%
618   {%
619     \ekvd@ifnoarg{#2}{\ekvd@err@missing@definition@msg{#1}}%
620   }
621 \long\def\ekvd@ifnoarg#1%
622   {%
623     \ekvd@ifnoarg@\ekvd@ifnoarg@mark#1\ekvd@ifnoarg@mark\ekvd@ifnoarg@t
624       \ekvd@ifnoarg@mark\ekvd@noarg@mark\ekvd@ifnoarg@mark\@secondoftwo
625   }
626 \long\def\ekvd@ifnoarg@#1\ekvd@ifnoarg@mark\ekvd@noarg@mark\ekvd@ifnoarg@mark{}%
627 \long\def\ekvd@ifnoarg@t
628   \ekvd@ifnoarg@mark\ekvd@noarg@mark\ekvd@ifnoarg@mark\@secondoftwo#1#2%
629   {%
630     #1%
631   }

```

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

\ekvd@assert@filledarg
 \ekvd@ifnoarg@or@empty 632 \long\def\ekvd@assert@filledarg#1%

```

633   {%
634     \ekvd@ifnoarg@or@empty{#1}\ekvd@err@missing@definition
635   }
636 \long\def\ekvd@ifnoarg@or@empty#1%
637   {%
638     \ekvd@ifnoarg{#1}%
639     \@\firstoftwo
640     {\ekv@ifempty{#1}}%
641   }

```

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

```

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

```

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.

```

642 \def\ekvd@assert@not@long{\ifx\ekvd@long\long\ekvd@err@no@prefix{long}\fi}
643 \def\ekvd@assert@not@protected
644   {%
645     \ifx\ekvd@prot\protected\ekvd@err@no@prefix{protected}\fi
646   }
647 \def\ekvd@assert@not@also{\ekvd@ifalso{\ekvd@err@no@prefix{also}}{}}
648 \def\ekvd@assert@not@long@also
649   {%
650     \ifx\ekvd@long\long\ekvd@err@no@prefix@also{long}\fi
651   }
652 \def\ekvd@assert@not@protected@also
653   {%
654     \ifx\ekvd@prot\protected\ekvd@err@no@prefix@also{protected}\fi
655   }

```

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

```

\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.

```

656 \long\def\ekvd@ifspace#1%
657   {%
658     \ekvd@ifspace@#1 \ekv@ifempty@B
659     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@\firstoftwo
660   }
661 \long\def\ekvd@ifspace@#1 % keep this space
662   {%
663     \ekv@ifempty@ \ekv@ifempty@A
664   }

```

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

2.2.5 Messages

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

```

\ekvd@errm
\ekvd@err@missing@definition
\ekvd@err@missing@definition@msg
\ekvd@err@missing@type
\ekvd@err@undefined@prefix
\ekvd@err@undefined@key
\ekvd@err@no@prefix
\ekvd@err@no@prefix@msg
\ekvd@err@no@prefix@also
\ekvd@err@add@val@on@noval
\ekvd@err@add@noval@on@val
\ekvd@err@unsupported@arg

```

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

```

665 \protected\def\ekvd@errm#1{\errmessage{\expkv@DEF Error: #1}}
666 \protected\def\ekvd@err@missing@definition

```

```

667   {\ekvd@errm{Missing definition for key '\ekvd@cur'}}
668   \protected\def\ekvd@err@missing@definition@msg#1%
669   {\ekvd@errm{Missing definition for key '\unexpanded{#1}'}}
670   \protected\def\ekvd@err@missing@type
671   {\ekvd@errm{Missing type prefix for key '\ekvd@cur'}}
672   \protected\def\ekvd@err@undefined@prefix#1%
673   {%
674     \ekvd@errm
675     {Undefined prefix '\unexpanded{#1}' found while processing '\ekvd@cur'}%
676   }
677   \protected\def\ekvd@err@undefined@key#1%
678   {%
679     \ekvd@errm
680     {Undefined key '\unexpanded{#1}' found while processing '\ekvd@cur'}%
681   }
682   \protected\def\ekvd@err@no@prefix#1%
683   {\ekvd@errm{prefix '#1' not accepted in '\ekvd@cur'}}
684   \protected\def\ekvd@err@no@prefix@msg#1#2%
685   {\ekvd@errm{prefix '#2' not accepted in '\unexpanded{#1}'}}
686   \protected\def\ekvd@err@no@prefix@also#1%
687   {\ekvd@errm{'\ekvd@cur' not allowed with a '#1' key}}
688   \protected\def\ekvd@err@add@val@on@noval
689   {\ekvd@errm{'\ekvd@cur' not allowed with a NoVal key}}
690   \protected\def\ekvd@err@add@noval@on@val
691   {\ekvd@errm{'\ekvd@cur' not allowed with a value taking key}}
692   \protected\def\ekvd@err@unsupported@arg\fi@firstofone#1%
693   {%
694     \fi
695     \ekvd@errm
696     {%
697       Existing key-macro has the unsupported argument string
698       '\ekvd@extracted@args' for key '\ekvd@cur'%
699     }%
700   }

```

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

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

The expandable error messages use \ekvd@err, which is just like \ekv@err from **expKV** or the way **expl3** throws expandable error messages. It uses an undefined control sequence to start the error message. \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.

```

701 \def\ekvd@err@choice@invalid#1%
702 {%
703   \ekvd@err@choice@invalid@#1\ekv@stop
704 }
705 \begingroup
706 \catcode40=8
707 \catcode41=8
708 \@firstofone{\endgroup
709 \def\ekvd@choice@name#1#2#3%
710 {%
711   \ekvd#1(#2)#3%

```

```

712     }
713 \def\ekvd@unknown@choice@name#1#2%
714   {%
715     \ekvd@u:#1(#2)%
716   }
717 \def\ekvd@err@choice@invalid@ \ekvd#1(#2)#3\ekv@stop%
718   {%
719     \ekv@ifdefined{\ekvd@unknown@choice@name{#1}{#2}}%
720       {\csname\ekvd@unknown@choice@name{#1}{#2}\endcsname{#3}}%
721       {\ekvd@err{invalid choice '#3' ('#2', set '#1')}}%
722   }
723 }
724 \begingroup
725 \edef\ekvd@err
726   {%
727     \endgroup
728     \unexpanded{\long\def\ekvd@err}##1%
729     {%
730       \unexpanded{\expandafter\ekv@err@\@firstofone}%
731       {\unexpanded{\expandafter{\csname ! expkv-def Error:\endcsname}##1.}}%
732       \unexpanded{\ekv@stop}%
733     }%
734   }
735 \ekvd@err

```

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

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

```
736 \catcode`@=\ekvd@tmp
```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

A	
also	3
apptoks	5
B	
bool	4
boolTF	4
box	5
C	
choice	6
code	3
D	
data	4
dataT	5
default	4
dimen	5
E	
ecode	3
edata	4
edataT	5
edefault	4
edimen	5
einitial	4
eint	5
\ekvchangeset	78, 79, 84, 85
\ekvdDate	2, 5, 9, 15
\ekvdef	226, 253, 270, 286, 303, 328, 343, 502
\ekvdefinekeys	2, 33
\ekvdefNoVal	84, 85, 201, 510
\ekvdVersion	2, 5, 9, 15
\ekvifdefined	118, 139, 155, 169, 183, 573, 595
\ekvifdefinedNoVal	576, 592
\ekvlet	109, 371, 389, 407
\ekvletNoVal	97, 128, 146, 376, 394
\ekvparses	36, 411
\ekvset	361
enoval	3
eskip	5
estore	4
G	
gapptoks	5
gbool	4
gboolTF	4
I	
gbox	4
gdata	5
gdataT	5
gdimen	5
gint	5
gskip	5
gstore	4
gtoks	5
L	
long	3
M	
meta	5
N	
nmeta	5
noval	3
O	
oinitial	4
P	
protect	3
protected	3
Q	
qdefault	4
S	
set	6
skip	5
smeta	5
snmeta	6
store	4
T	
TeX and L ^A T _E X 2 _E commands:	
\ekv@err@	730
\ekv@fi@gobble	555, 559
\ekv@gobble@mark	455
\ekv@ifdefined	50, 53, 437, 719
\ekv@ifempty	72, 640
\ekv@ifempty@	570, 663
\ekv@ifempty@A	568, 570, 659, 663

\ekv@ifempty@B 567, 568, 658, 659
 \ekv@ifempty@false 568, 659
 \ekv@mark 44, 47, 423, 433
 \ekv@name 124,
 143, 160, 174, 189, 501, 509, 574, 593
 \ekv@stop 44,
 48, 61, 423, 435, 441, 703, 717, 732
 \ekv@strip 47, 51, 433, 445
 \ekvd@ 36, 38
 \ekvd@add@aux 497
 \ekvd@add@aux@ 497
 \ekvd@add@noval
 ... 78, 79, 96, 127, 145, 376, 394, 497
 \ekvd@add@val .. 108, 223, 250, 268,
 283, 301, 326, 341, 371, 389, 405, 497
 \ekvd@assert@arg 91, 104,
 116, 137, 153, 167, 181, 466, 475, 616
 \ekvd@assert@arg@msg 420, 616
 \ekvd@assert@filledarg .. 197, 217,
 244, 262, 277, 295, 320, 335, 350, 632
 \ekvd@assert@not@also
 ... 157, 171, 185, 478, 647
 \ekvd@assert@not@also\uuuuu\ekvd@assert@not@long@also
 ... 642
 \ekvd@assert@not@long
 ... 70, 93, 120, 141,
 158, 172, 186, 374, 392, 398, 477, 642
 \ekvd@assert@not@long@also
 ... 376, 394, 405
 \ekvd@assert@not@protected
 ... 71, 159, 173, 187, 642
 \ekvd@assert@not@protected@also
 ... 80, 642
 \ekvd@assert@noval 507, 571
 \ekvd@assert@noval@ 571
 \ekvd@assert@twoargs 380, 561
 \ekvd@assert@val 499, 571
 \ekvd@assert@val@ 571
 \ekvd@choice@invalid@p .. 453, 462, 463
 \ekvd@choice@invalid@p@ ... 455, 458
 \ekvd@choice@name
 ... 203, 206, 401, 426, 445, 701
 \ekvd@choice@p@also 463
 \ekvd@choice@p@long 396
 \ekvd@choice@p@long@ 396
 \ekvd@choice@p@protect 396
 \ekvd@choice@p@protected 396
 \ekvd@choice@prefix 396
 \ekvd@choice@prefix@ 396
 \ekvd@clear@prefixes 20, 41, 419
 \ekvd@cur 42, 415, 420, 460, 667,
 671, 675, 680, 683, 687, 689, 691, 698
 \ekvd@empty 20, 322, 337, 604
 \ekvd@err 701
 \ekvd@err@add@noval@on@val .. 596, 665
 \ekvd@err@add@val@on@noval .. 577, 665
 \ekvd@err@choice@invalid .. 492, 701
 \ekvd@err@choice@invalid@ .. 701
 \ekvd@err@missing@definition ...
 ... 73, 563, 616, 634, 665
 \ekvd@err@missing@definition@msg
 ... 415, 619, 665
 \ekvd@err@missing@type .. 45, 62, 665
 \ekvd@err@no@prefix .. 642, 645, 647, 665
 \ekvd@err@no@prefix@also .. 650, 654, 665
 \ekvd@err@no@prefix@msg .. 460, 665
 \ekvd@err@undefined@key
 ... 130, 148, 162, 176, 192, 578, 597, 665
 \ekvd@err@undefined@prefix
 ... 55, 449, 665
 \ekvd@err@unsupported@arg
 ... 586, 605, 665
 \ekvd@errm 665
 \ekvd@exp@Nno
 ... 28, 96, 108, 127, 145, 355, 384, 405
 \ekvd@exp@reinsert@n 28
 \ekvd@extract@args 571
 \ekvd@extract@prefixes .. 515, 523
 \ekvd@extract@prefixes@ 523
 \ekvd@extract@prefixes@long .. 523
 \ekvd@extract@prefixes@prot .. 523
 \ekvd@extracted@args 571, 698
 \ekvd@h@choice 401, 483
 \ekvd@h@choice@ 483
 \ekvd@ifalso 20, 67,
 75, 95, 107, 126, 144, 220, 247, 265,
 280, 298, 323, 338, 354, 383, 403, 647
 \ekvd@ifempty@gtwo 561
 \ekvd@ifnoarg 77, 83, 616, 638
 \ekvd@ifnoarg@ 616
 \ekvd@ifnoarg@mark .. 623, 624, 626, 628
 \ekvd@ifnoarg@or@empty 632
 \ekvd@ifnoarg@t 623, 627
 \ekvd@ifnottwoargs 561
 \ekvd@ifspace
 ... 43, 60, 422, 440, 453, 456, 656
 \ekvd@ifspace@ 656
 \ekvd@long
 ... 20, 64, 106, 226, 253, 270, 286,
 303, 328, 343, 369, 502, 533, 642, 650

\ekvd@mark	532, 535, 538, 540	\ekvd@t@estore	333
\ekvd@newlet	199, 219, 322, 337, 553	\ekvd@t@gapptoks	275
\ekvd@newreg	246, 264, 279, 297, 553	\ekvd@t@gbool	195
\ekvd@noarg	36, 38	\ekvd@t@gboolTF	195
\ekvd@noarg@mark	38, 624, 626, 628	\ekvd@t@gbox	242
\ekvd@one@arg@string	571	\ekvd@t@gdata	215
\ekvd@p@also	64	\ekvd@t@gdataT	215
\ekvd@p@long	64	\ekvd@t@gdimen	293
\ekvd@p@protect	64	\ekvd@t@gint	293
\ekvd@p@protected	64	\ekvd@t@gskip	293
\ekvd@populate@choice	396	\ekvd@t@gstore	318
\ekvd@populate@choice@	396	\ekvd@t@gtoks	260
\ekvd@populate@choice@noarg	396	\ekvd@t@initial	151
\ekvd@prefix	44, 47, 61	\ekvd@t@int	293
\ekvd@prefix@	47	\ekvd@t@meta	348
\ekvd@prefix@after@p	54, 58	\ekvd@t@nmeta	348
\ekvd@prot	20, 65, 94, 106, 121, 142, 222, 249, 267, 282, 325, 340, 369, 399, 443, 451, 479, 521, 536, 645, 654	\ekvd@t@noval	89
\ekvd@set	35, 84, 85, 97, 109, 118, 124, 128, 139, 143, 146, 155, 160, 169, 174, 183, 189, 201, 203, 206, 226, 253, 270, 286, 303, 328, 343, 352, 356, 385, 401, 407, 426, 445, 480, 501, 509, 521, 573, 574, 576, 592, 593, 595	\ekvd@t@oinital	151
\ekvd@set@choice	426, 445, 469	\ekvd@t@qdefault	114
\ekvd@stop	525, 529, 532, 535, 538, 540, 584, 603, 611	\ekvd@t@set	68
\ekvd@t@gapptoks	275	\ekvd@t@skip	293
\ekvd@t@bool	195	\ekvd@t@smeta	378
\ekvd@t@boolTF	195	\ekvd@t@snmeta	378
\ekvd@t@box	242	\ekvd@t@store	318
\ekvd@t@choice	396	\ekvd@t@toks	260
\ekvd@t@code	102	\ekvd@t@unknown-choice	473
\ekvd@t@data	215	\ekvd@t@xdata	236
\ekvd@t@dataT	215	\ekvd@t@xdataT	241
\ekvd@t@default	114	\ekvd@t@xdimen	293
\ekvd@t@dimen	293	\ekvd@t@xint	293
\ekvd@t@ecode	102	\ekvd@t@xskip	293
\ekvd@t@edata	233	\ekvd@t@xstore	333
\ekvd@t@edataT	238	\ekvd@tmp	2, 94, 96, 97, 106, 108, 109, 121, 127, 128, 142, 145, 146, 188, 190, 352, 353, 355, 356, 369, 384, 385, 399, 405, 407, 544, 552, 736
\ekvd@t@edefault	135	\ekvd@type@gapptoks	275
\ekvd@t@edimen	293	\ekvd@type@bool	195
\ekvd@t@einitial	151	\ekvd@type@box	242
\ekvd@t@eint	293	\ekvd@type@choice	200, 396
\ekvd@t@enoval	89	\ekvd@type@code	102
\ekvd@t@eskip	293	\ekvd@type@data	215

\ekvd@type@meta@c	<u>348</u>	U
\ekvd@type@noval	<u>89</u>	unknown-choice
\ekvd@type@reg	<u>293</u>	
\ekvd@type@smeta	<u>378</u>	X
\ekvd@type@smeta@	<u>378</u>	xdata
\ekvd@type@store	<u>318</u>	xdataT
\ekvd@type@toks	<u>260</u>	xdimen
\ekvd@unknown@choice@name	<u>480, 701</u>	xint
\evkd@prot	<u>300</u>	xskip
toks	<u>5</u>	xstore