

exp_k^v|DEF

a key-defining frontend for exp_k^v

Jonathan P. Spratte*

2021-09-18 vo.8b

Abstract

exp_k^v|DEF provides a small $\langle key \rangle = \langle value \rangle$ interface to define keys for exp_k^v. Key-types are declared using prefixes, similar to static typed languages. The stylised name is exp_k^v|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	8
1.4	Example	8
1.5	License	9
2	Implementation	10
2.1	The L ^A T _E X Package	10
2.2	The Generic Code	10
2.2.1	Key Types	12
2.2.2	Key Type Helpers	25
2.2.3	Handling also	25
2.2.4	Tests	26
2.2.5	Messages	29
	Index	32

*jspratte@yahoo.de

1 Documentation

Since the trend for the last couple of years goes to defining keys for a $\langle key \rangle = \langle value \rangle$ interface using a $\langle key \rangle = \langle value \rangle$ interface, I thought that maybe providing such an interface for `expkv` will make it more attractive for actual use, besides its unique selling points of being fully expandable, and fast and reliable. But at the same time I don't want to widen `expkv`'s initial scope. So here it is `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 [subsection 1.2.1](#), the t-prefixes are described in [subsection 1.2.2](#).

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

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

1.1 Macros

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

<code>\ekvdefinekeys</code>	<code>\ekvdefinekeys{<set>}{<key>=<value>, ...}</code>
-----------------------------	--

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.

<code>\ekvdDate</code> <code>\ekvdVersion</code>	
---	--

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 $\langle value \rangle$. 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 $\langle key \rangle$'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.

<u>code</u> <u>ecode</u>	<code>code <key> = {\<definition>}</code> new 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.
<u>noval</u> <u>enoval</u>	<code>noval <key> = {\<definition>}</code> new 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.
<u>default</u> <u>qdefault</u> <u>odefault</u> <u>fdefault</u> <u>edefault</u>	<code>default <key> = {\<definition>}</code> new 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>. odefault is just another name for qdefault. The fdefault version will expand the key code until a non-expandable token or a space is found, a space would be gobbled. ¹ The edefault on the other hand fully expands the <key>-code with <definition> as its argument inside of an \edef.
<u>initial</u> <u>oinitial</u> <u>finitial</u> <u>einitial</u>	<code>initial <key> = {\<value>}</code> new 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. finitial will expand <value> until a non-expandable token or a space is found, a space would be gobbled. ²
	If you don't provide a value (and no equals sign) a noval <key> of the same name is called once (or, if you specified a default for a value taking key that would be used).

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

²Again using \romannumeral.

<hr/> bool gbool boolTF gboolTF <hr/>	<p>bool $\langle key \rangle = \langle cs \rangle$ new also protected long</p> <p>The $\langle cs \rangle$ should be a single control sequence, such as <code>\iff foo</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 <code>T_EX-ifs</code> so either be <code>\if true</code> or <code>\if false</code>. The <code>boolTF</code> and <code>gboolTF</code> variants will both take two arguments and if <code>true</code> 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.</p>
<hr/> invbool ginvbool invboolTF ginvboolTF <hr/>	<p>bool $\langle key \rangle = \langle cs \rangle$ new also protected long</p> <p>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>\if false</code> and vice versa.</p>
<hr/> boolpair gboolpair boolpairTF gboolpairTF <hr/>	<p>boolpair $\langle key \rangle = \langle cs_1 \rangle \langle cs_2 \rangle$ new also protected long</p> <p>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.</p>
<hr/> store estore gstore xstore <hr/>	<p>store $\langle key \rangle = \langle cs \rangle$ new also protected long</p> <p>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>.</p>
<hr/> data edata gdata xdata <hr/>	<p>data $\langle key \rangle = \langle cs \rangle$ new also protected long</p> <p>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>.</p>
<hr/> dataT edataT gdataT xdataT <hr/>	<p>dataT $\langle key \rangle = \langle cs \rangle$ new also protected long</p> <p>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>.</p>

<u>int</u> eint gint xint	int $\langle key \rangle = \langle cs \rangle$ The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . An int key will be a TeX-count register. If $\langle cs \rangle$ isn't defined yet, <code>\newcount</code> will be used to initialise it. The eint and xint versions will use <code>\numexpr</code> to allow basic computations in their $\langle value \rangle$. The gint and xint variants set the register globally.	new also protected long
<u>dimen</u> edimen gdimen xdimen	dimen $\langle key \rangle = \langle cs \rangle$ The $\langle cs \rangle$ 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.	new also protected long
<u>skip</u> eskip gskip xskip	skip $\langle key \rangle = \langle cs \rangle$ The $\langle cs \rangle$ 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.	new also protected long
<u>toks</u> gtoks apptoks gapptoks	toks $\langle key \rangle = \langle cs \rangle$ The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . Store $\langle value \rangle$ inside of a toks-register. The g variants use <code>\global</code> , 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 <code>\newtoks</code> .	new also protected long
<u>box</u> gbox	box $\langle key \rangle = \langle cs \rangle$ The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . Typesets $\langle value \rangle$ into a <code>\hbox</code> and stores the result in a box register. The boxes are colour safe. <code>exp\KVIDEF</code> doesn't provide a vbox type.	new also protected long
<u>meta</u>	meta $\langle key \rangle = \{\langle key \rangle = \langle value \rangle, \dots\}$ 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- <code>\ekvset</code> 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 <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.	new also protected long
<u>nmeta</u>	nmeta $\langle key \rangle = \{\langle key \rangle = \langle value \rangle, \dots\}$ 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.	new also protected long
<u>smeta</u>	smeta $\langle key \rangle = \{\langle set \rangle\}\{\langle key \rangle = \langle value \rangle, \dots\}$ 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 <code>\ekvset{\langle set \rangle}\{\langle key \rangle = \langle value \rangle, \dots\}</code> .	new also protected long
<u>snmeta</u>	snmeta $\langle key \rangle = \{\langle set \rangle\}\{\langle key \rangle = \langle value \rangle, \dots\}$ 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 $\langle set \rangle$.	new also protected long

set	<code>set <key> = {<set>}</code>	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	<code>choice <key> = {<value>=<definition>, ...}</code>	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	<code>unknown-choice <key> = {<definition>}</code>	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	<code>unknown code = {<definition>}</code>	new also protected long
---------------------	--	-------------------------

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

With `unknown code` the `<definition>` is used for unknown keys which were provided a value (so corresponds to `\ekvdefunknown`), you can access the key name with `#1` and the value with `#2`.³

unknown_noval	<code>unknown noval = {<definition>}</code>	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	<code>unknown redirect-code = {<set-list>}</code>	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.

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

<code>unknown_redirect-noval</code>	<code>unknown redirect-noval = {\set-list}</code>	<code>new also protected long</code>
-------------------------------------	---	--------------------------------------

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.

<code>unknown_redirect</code>	<code>unknown redirect = {\set-list}</code>	<code>new also protected long</code>
-------------------------------	---	--------------------------------------

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

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 L^AT_EX Project Public License (LPPL), either version 1.3c of this license or (at your option) any later version. The latest version of this license is in the file:

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

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

2 Implementation

2.1 The L^AT_EX Package

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

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

2.2 The Generic Code

The rest of this implementation will be the generic code.

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

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

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

```
\ekvdDate
15 \def\ekvdVersion{0.8b}
16 \def\ekvdDate{2021-09-18}
```

(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 `expkvDEF` was initialized.

```
\ekvd@long \expkvDEF will use \ekvd@long, \ekvd@prot, and \ekvd@ifalso to store whether a key
\ekvd@prot should be defined as \long or \protected or adds an action to an existing key, and we
\ekvd@clear@prefixes have to clear them for every new key. By default long and protected will just be empty,
\ekvd@empty ifalso will be \@secondoftwo, and ifnew will just use its third argument.
\ekvd@ifalso
20 \def\ekvd@empty{}
```

```

21 \protected\def\ekvd@clear@prefixes
22 {%
23   \let\ekvd@long\ekvd@empty
24   \let\ekvd@prot\ekvd@empty
25   \let\ekvd@ifalso\@secondoftwo
26   \long\def\ekvd@ifnew##1##2##3{##3}%
27 }
28 \ekvd@clear@prefixes

```

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

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

```

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

```

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

\ekvd@noarg \ekvd@noarg and \ekvd@arg store whether there was a value in the $\langle key \rangle = \langle value \rangle$ pair.
 \ekvd@arg \ekvd@handle has to test whether there is a space inside the key and if so calls the prefix
 \ekvd@handle grabbing routine, else we throw an error and ignore the key.

```

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

```

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

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

```

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

```

```

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

```

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

\ekvd@prefix@after@p The @p@ type prefixes are all just modifying a following @t@ type, so they will need to search for another prefix. This is true for all of them, so we use a macro to handle this. It'll throw an error if there is no other prefix.

```

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

```

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

\ekvd@p@long
\ekvd@p@protected
\ekvd@p@protect
\ekvd@p@also
\ekvd@p@new

Define the @p@ type prefixes, they all just store some information in a temporary macro.

```

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

```

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

2.2.1 Key Types

\ekvd@type@set
\ekvd@t@set

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

```

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

```

```

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

```

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

\ekvd@type@noval Another pretty simple type, noval just needs to assert that there is a definition and that 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.

```

99 \protected\long\def\ekvd@type@noval#1#2#3%
100 {%
101   \ekvd@ifnew{NoVal}{#2}%
102   {%
103     \ekvd@assert@arg
104     {%
105       \ekvd@assert@not@long
106       \ekvd@prot#1\ekvd@tmp{#3}%
107       \ekvd@ifalso
108         {\ekv@exparg{\ekvd@add@noval{#2}}\ekvd@tmp{}}%
109         {\ekvletNoVal\ekvd@set{#2}\ekvd@tmp}%
110       }%
111     }%
112   }
113 \protected\def\ekvd@t@noval{\ekvd@type@noval\def}
114 \protected\def\ekvd@t@enoval{\ekvd@type@noval\edef}

```

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

\ekvd@type@code code is simple as well, ecode has to use \edef on a temporary macro, since **explv** doesn't provide an \ekvedef.

```

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

```

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

`\ekvd@type@default` `\ekvd@type@default` asserts there was an argument, also the key for which one wants to set a default has to be already defined (this is not so important for `default`, but `qdefault` requires is). If everything is good, `\edef` a temporary macro that expands `\ekvd@set` and the `\csname` for the key, and in the case of `qdefault` does the first expansion step of the key-macro.

```

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

```

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

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

```

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

```

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

```

\ekvd@t@initial
\ekvd@t@oinitial 171 \long\def\ekvd@type@initial#1#2#3#4%
\ekvd@t@finitial 172 {%
\ekvd@t@einitial 173 \ekvd@assert@not@new
174 \ekvd@assert@not@also
175 \ekvd@assert@not@long
176 \ekvd@assert@not@protected
177 \ekvd@ifnoarg
178 {%
179 \ekvifdefinedNoVal\ekvd@set{#3}%
180 {\csname\ekv@name\ekvd@set{#3}N\endcsname}%
181 {\ekvd@err@undefined@noval{#3}}%
182 }%
183 {%
184 \ekvifdefined\ekvd@set{#3}%
185 {%
186 #1{#2#4}%
187 \csname\ekv@name\ekvd@set{#3}\expandafter\endcsname\expandafter
188 {\ekvd@tmp}%
189 }%
190 {\ekvd@err@undefined@key{#3}}%
191 }%
192 }
193 \def\ekvd@t@initial{\ekvd@type@initial{\def\ekvd@tmp{}}}
194 \def\ekvd@t@oinitial{\ekvd@type@initial{\ekv@exparg{\def\ekvd@tmp{}}{}}}
195 \def\ekvd@t@einitial{\ekvd@type@initial{\edef\ekvd@tmp{}}}
196 \def\ekvd@t@finitial
197 {\ekvd@type@initial{\ekv@exparg{\def\ekvd@tmp{}}{\romannumeral'^^@}}}

```

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

<pre> \ekvd@type@bool \ekvd@t@bool \ekvd@t@gbool \ekvd@t@boolTF \ekvd@t@gboolTF \ekvd@t@invbool \ekvd@t@ginvbool \ekvd@t@invboolTF \ekvd@t@ginvboolTF </pre>	<p>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).</p> <pre> 198 \protected\def\ekvd@type@bool#1#2#3#4#5% 199 {% 200 \ekvd@ifnew{#4}% 201 {% 202 \ekvd@ifnew{NoVal}{#4}% 203 {% 204 \ekvd@assert@filledarg{#5}% 205 {% 206 \ekvd@newlet#5#3% 207 \ekvd@type@choice{#4}% 208 \protected\ekvdefNoVal\ekvd@set{#4}{#1\let#5#2}% 209 \protected\expandafter\def 210 \csname\ekvd@choice@name\ekvd@set{#4}{true}\endcsname 211 {#1\let#5#2}% 212 \protected\expandafter\def 213 \csname\ekvd@choice@name\ekvd@set{#4}{false}\endcsname 214 {#1\let#5#3}% 215 }% 216 }% 217 }% </pre>
--	---

```

218 }
219 \protected\def\ekvd@t@bool{\ekvd@type@bool{}\iftrue\iffalse}
220 \protected\def\ekvd@t@gbool{\ekvd@type@bool\global\iftrue\iffalse}
221 \protected\def\ekvd@t@boolTF{\ekvd@type@bool{}\@firstoftwo\@secondoftwo}
222 \protected\def\ekvd@t@gboolTF{\ekvd@type@bool\global\@firstoftwo\@secondoftwo}
223 \protected\def\ekvd@t@invbool{\ekvd@type@bool{}\iffalse\iftrue}
224 \protected\def\ekvd@t@ginvbool{\ekvd@type@bool\global\iffalse\iftrue}
225 \protected\def\ekvd@t@invboolTF{\ekvd@type@bool{}\@secondoftwo\@firstoftwo}
226 \protected\def\ekvd@t@ginvboolTF
227 {\ekvd@type@bool\global\@secondoftwo\@firstoftwo}

```

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

The boolean pair types are essentially the same as the boolean types, but set two macros instead of one.

```

\ekvd@type@boolpair
\ekvd@t@boolpair
\ekvd@t@gboolpair
\ekvd@t@boolpairTF
\ekvd@t@gboolpairTF
228 \protected\def\ekvd@type@boolpair#1#2#3#4#5#6%
229 {%
230   \ekvd@ifnew{}\{#4}%
231   {%
232     \ekvd@ifnew{NoVal}\{#4}%
233     {%
234       \ekvd@newlet#5#3%
235       \ekvd@newlet#6#2%
236       \ekvd@type@choice{#4}%
237       \protected\ekvdefNoVal\ekvd@set{#4}\{#1\let#5#2#1\let#6#3}%
238       \protected\expandafter\def
239         \csname\ekvd@choice@name\ekvd@set{#4}\{true}\endcsname
240         {#1\let#5#2#1\let#6#3}%
241       \protected\expandafter\def
242         \csname\ekvd@choice@name\ekvd@set{#4}\{false}\endcsname
243         {#1\let#5#3#1\let#6#2}%
244     }%
245   }%
246 }
247 \protected\def\ekvd@t@boolpair#1#2%
248 {\ekvd@assert@twoargs{#2}\{ekvd@type@boolpair{}\iftrue\iffalse{#1}#2}}
249 \protected\def\ekvd@t@gboolpair#1#2%
250 {\ekvd@assert@twoargs{#2}\{ekvd@type@boolpair\global\iftrue\iffalse{#1}#2}}
251 \protected\def\ekvd@t@boolpairTF#1#2%
252 {%
253   \ekvd@assert@twoargs{#2}%
254   {\ekvd@type@boolpair{}\@firstoftwo\@secondoftwo{#1}#2}%
255 }
256 \protected\def\ekvd@t@gboolpairTF#1#2%
257 {%
258   \ekvd@assert@twoargs{#2}%
259   {\ekvd@type@boolpair\global\@firstoftwo\@secondoftwo{#1}#2}%
260 }

```

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

```

\ekvd@type@data
\ekvd@t@data
\ekvd@t@gdata
\ekvd@t@dataT
\ekvd@t@gdataT
261 \protected\def\ekvd@type@data#1#2#3#4#5#6%
262 {%

```



```

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

```

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

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

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

\ekvd@t@gbox

```

290 \protected\def\ekvd@type@box#1#2#3%
291 {%
292 \ekvd@ifnew{ }{#2}%
293 {%
294 \ekvd@assert@filledarg{#3}%
295 {%
296 \ekvd@newreg#3{box}%
297 \ekvd@ifalso
298 {%
299 \let\ekvd@prot\protected
300 \ekvd@add@val{#2}{#1\setbox#3\hbox{\begingroup##1\endgroup}}{ }%
301 }%
302 {%
303 \protected\ekvd@long\ekvdef\ekvd@set{#2}%
304 {#1\setbox#3\hbox{\begingroup##1\endgroup}}%
305 }%
306 }%
307 }%
308 }
309 \protected\def\ekvd@t@box{\ekvd@type@box}

```

```
310 \protected\def\ekvd@t@gbbox{\ekvd@type@box\global}
```

(End definition for \ekvd@type@box, \ekvd@t@box, and \ekvd@t@gbbox.)

\ekvd@type@toks Similar to box, but set the toks.

```
\ekvd@t@toks 311 \protected\def\ekvd@type@toks#1#2#3%
\ekvd@t@gtoks 312 {%
313 \ekvd@ifnew{#2}%
314 {%
315 \ekvd@assert@filledarg{#3}%
316 {%
317 \ekvd@newreg#3{toks}%
318 \ekvd@ifalso
319 {%
320 \let\ekvd@prot\protected
321 \ekvd@add@val{#2}{#1#3{##1}}{}%
322 }%
323 {\protected\ekvd@long\ekvdef\ekvd@set{#2}{#1#3{##1}}}%
324 }%
325 }%
326 }
327 \protected\def\ekvd@t@toks{\ekvd@type@toks{}}
328 \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 329 \protected\def\ekvd@type@apptoks#1#2#3%
330 {%
331 \ekvd@ifnew{#2}%
332 {%
333 \ekvd@assert@filledarg{#3}%
334 {%
335 \ekvd@newreg#3{toks}%
336 \ekvd@ifalso
337 {%
338 \let\ekvd@prot\protected
339 \ekvd@add@val{#2}{#1#3\expandafter{\the#3##1}}{}%
340 }%
341 {%
342 \protected\ekvd@long\ekvdef\ekvd@set{#2}%
343 {#1#3\expandafter{\the#3##1}}%
344 }%
345 }%
346 }%
347 }
348 \protected\def\ekvd@t@apptoks{\ekvd@type@apptoks{}}
349 \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 350 \protected\def\ekvd@type@reg#1#2#3#4#5#6%
```

```
\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
```

```

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

```

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

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

```

\ekvd@t@gstore
378   \protected\def\ekvd@type@store#1#2#3#4%
379   {%
380     \ekvd@ifnew{ }{#3}%
381     {%
382       \ekvd@assert@filledarg{#4}%
383       {%
384         \ekvd@newlet#4\ekvd@empty
385         \ekvd@ifalso
386         {%
387           \let\ekvd@prot\protected
388           \ekvd@add@val{#3}{#1#4{#2}}{ }%
389         }%
390         {\protected\ekvd@long\ekvdef\ekvd@set{#3}{#1#4{#2}}}%
391       }%
392     }%
393   }
394   \protected\def\ekvd@t@store{\ekvd@type@store\edef{\unexpanded{##1}}}
395   \protected\def\ekvd@t@gstore{\ekvd@type@store\xdef{\unexpanded{##1}}}
396   \protected\def\ekvd@t@estore{\ekvd@type@store\edef{##1}}
397   \protected\def\ekvd@t@xstore{\ekvd@type@store\xdef{##1}}

```

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

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

```

\ekvd@type@meta@a
\ekvd@type@meta@b
\ekvd@type@meta@c
\ekvd@t@meta
\ekvd@t@nmeta
398 \protected\long\def\ekvd@type@meta#1#2#3#4#5#6#7%
399 {%
400   \ekvd@ifnew{#1}{#6}%
401   {%
402     \ekvd@assert@filledarg{#7}%
403     {%
404       \edef\ekvd@tmp{\ekvd@set}%
405       \expandafter\ekvd@type@meta@a\expandafter{\ekvd@tmp}{#7}{#2}%
406       \ekvd@ifalso
407         {\ekv@exparg{#3{#6}}{\ekvd@tmp#4}{#5}}%
408         {\csname ekvlet#1\endcsname\ekvd@set{#6}\ekvd@tmp}%
409     }%
410   }%
411 }
412 \protected\long\def\ekvd@type@meta@a#1#2%
413 {%
414   \expandafter\ekvd@type@meta@b\expandafter{\ekvset{#1}{#2}}%
415 }
416 \protected\def\ekvd@type@meta@b
417 {%
418   \expandafter\ekvd@type@meta@c\expandafter
419 }
420 \protected\long\def\ekvd@type@meta@c#1#2%
421 {%
422   \ekvd@prot\ekvd@long\def\ekvd@tmp#2{#1}%
423 }
424 \protected\def\ekvd@t@meta{\ekvd@type@meta}{##1}\ekvd@add@val{##1}{}}
425 \protected\def\ekvd@t@nmeta
426 {%
427   \ekvd@assert@not@long
428   \ekvd@type@meta{NoVal}{}\ekvd@add@noval{}\ekvd@assert@not@long@also
429 }

```

(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.

```

\ekvd@t@smeta
\ekvd@t@snmeta
430 \protected\long\def\ekvd@type@smeta#1#2#3#4#5#6#7%
431 {%
432   \ekvd@ifnew{#1}{#6}%
433   {%
434     \ekvd@assert@twoargs{#7}%
435     {%
436       \ekvd@type@meta@a#7{#2}%
437       \ekvd@ifalso
438         {\ekv@exparg{#3{#6}}{\ekvd@tmp#4}{#5}}%
439         {\csname ekvlet#1\endcsname\ekvd@set{#6}\ekvd@tmp}%
440     }%
441   }%
442 }
443 \protected\def\ekvd@t@smeta{\ekvd@type@smeta}{##1}\ekvd@add@val{##1}{}}
444 \protected\def\ekvd@t@snmeta

```

```

445 {%
446   \ekvd@assert@not@long
447   \ekvd@type@smeta{NoVal}{}\ekvd@add@noval{}\ekvd@assert@not@long@also
448 }

```

(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
449 \protected\def\ekvd@type@choice#1%
450 {%
451   \ekvd@assert@not@long
452   \ekvd@prot\edef\ekvd@tmp##1%
453   {\unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#1}{##1}}}%
454   \ekvd@ifalso
455   {%
456     \ekvd@assert@val{#1}%
457     {%
458       \ekvd@if@not@already@choice{#1}%
459       {%
460         \ekv@exparg
461         {%
462           \expandafter\ekvd@add@aux
463           \csname\ekv@name\ekvd@set{#1}\endcsname{##1}{#1}%
464         }%
465         {\ekvd@tmp{##1}}%
466         {\ekvd@long\ekvdef}\ekvd@assert@not@long@also
467       }%
468     }%
469   }%
470   {\ekvlet\ekvd@set{#1}\ekvd@tmp}%
471 }

```

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

```

472 \protected\def\ekvd@populate@choice
473 {%
474   \ekvparse\ekvd@populate@choice@noarg\ekvd@populate@choice@
475 }
476 \protected\long\def\ekvd@populate@choice@noarg#1%
477 {%
478   \expandafter\ekvd@err@missing@definition@msg\expandafter{\ekvd@cur : #1}%
479 }

```

\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.

```

480 \protected\long\def\ekvd@populate@choice@#1#2%
481 {%
482   \ekvd@clear@prefixes

```

```

483 \expandafter\ekvd@assert@arg@msg\expandafter{\ekvd@cur : #1}%
484 {%
485     \ekvd@ifspace{#1}%
486     {\ekvd@choice@prefix\ekv@mark#1\ekv@stop}%
487     {%
488         \expandafter\def
489         \csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname
490     }%
491     {#2}%
492 }%
493 }
494 \protected\def\ekvd@choice@prefix#1
495 {%
496     \ekv@strip{#1}\ekvd@choice@prefix@\ekv@mark
497 }
498 \protected\def\ekvd@choice@prefix@#1#2\ekv@stop
499 {%
500     \ekv@ifdefined{\ekvd@choice@p@#1}%
501     {%
502         \csname ekvd@choice@p@#1\endcsname
503         \ekvd@ifspace{#2}%
504         {\ekvd@choice@prefix#2\ekv@stop}%
505         {%
506             \ekvd@prot\expandafter\def
507             \csname
508                 \ekv@strip{#2}{\ekvd@choice@name\ekvd@set\ekvd@set@choice}%
509             \endcsname
510         }%
511     }%
512     {\ekvd@err@undefined@prefix{#1}\@gobble}%
513 }
514 \protected\def\ekvd@choice@p@protected{\let\ekvd@prot\protected}
515 \let\ekvd@choice@p@protect\ekvd@choice@p@protected
516 \protected\def\ekvd@choice@invalid@p#1\ekvd@ifspace#2%
517 {%
518     \expandafter\ekvd@choice@invalid@p@\expandafter{\ekv@gobble@mark#2}{#1}%
519     \ekvd@ifspace{#2}%
520 }
521 \protected\def\ekvd@choice@invalid@p@#1#2%
522 {%
523     \expandafter\ekvd@err@no@prefix@msg\expandafter{\ekvd@cur : #2 #1}{#2}%
524 }
525 \protected\def\ekvd@choice@p@long{\ekvd@choice@invalid@p@long}%
526 \protected\def\ekvd@choice@p@also{\ekvd@choice@invalid@p@also}%
527 \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.

```

528 \protected\long\def\ekvd@t@choice#1#2%
529 {%
530     \ekvd@ifnew{}{#1}%
531     {%
532         \ekvd@assert@arg
533         {%

```

```

534         \ekvd@type@choice{#1}%
535         \def\ekvd@set@choice{#1}%
536         \ekvd@populate@choice{#2}%
537     }%
538 }%
539 }

```

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

\ekvd@t@unknown-choice

```

540 \protected\long\expandafter\def\csname ekvd@t@unknown-choice\endcsname#1#2%
541 {%
542     \ekvd@assert@new@for@name{\ekvd@unknown@choice@name\ekvd@set{#1}}%
543     {%
544         \ekvd@assert@arg
545         {%
546             \ekvd@assert@not@long
547             \ekvd@assert@not@also
548             \ekvd@prot\expandafter
549             \def\csname\ekvd@unknown@choice@name\ekvd@set{#1}\endcsname##1{#2}%
550         }%
551     }%
552 }

```

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

\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.

```

553 \protected\long\def\ekvd@t@unknown#1#2%
554 {%
555     \ekv@ifdefined{ekvd@type@unknown@\detokenize{#1}}%
556     {\csname ekvd@type@unknown@\detokenize{#1}\endcsname{#2}}%
557     \ekvd@err@misused@unknown
558 }

```

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

```

559 \protected\long\def\ekvd@type@unknown@noval#1%
560 {%
561     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}}uN}%
562     {%
563         \ekvd@assert@arg
564         {%
565             \ekvd@assert@not@also
566             \ekvd@assert@not@long
567             \ekvd@prot\ekvdefunknownNoVal\ekvd@set{#1}%
568         }%
569     }%
570 }

```

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.

```

571 \protected\long\def\ekvd@type@unknown@code#1%

```

```

572   {%
573     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}}u}%
574     {%
575       \ekvd@assert@arg
576       {%
577         \ekvd@assert@not@also
578         \beginngroup
579         \def\ekvd@tmp##1##2{#1}%
580         \ekv@exparg
581         {%
582           \endgroup
583           \ekvd@prot\ekvd@long\ekvdefunknown\ekvd@set
584         }%
585         {\ekvd@tmp{##2}{##1}}%
586       }%
587     }%
588   }

```

(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.

```

589 \protected\edef\ekvd@type@unknown@redirect#1%
590   {%
591     \expandafter\noexpand\csname ekvd@type@unknown@redirect-code\endcsname{#1}%
592     \expandafter\noexpand\csname ekvd@type@unknown@redirect-noval\endcsname{#1}%
593   }
594 \protected\expandafter\def\csname ekvd@type@unknown@redirect-code\endcsname#1%
595   {%
596     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}}u}%
597     {%
598       \ekvd@assert@arg
599       {%
600         \ekvd@assert@not@also
601         \ekvd@assert@not@protected
602         \expandafter\ekvredirectunknown\expandafter{\ekvd@set}{#1}%
603       }%
604     }%
605   }
606 \protected\expandafter\def\csname ekvd@type@unknown@redirect-noval\endcsname#1%
607   {%
608     \ekvd@assert@new@for@name{\ekv@name\ekvd@set{}}uN}%
609     {%
610       \ekvd@assert@arg
611       {%
612         \ekvd@assert@not@also
613         \ekvd@assert@not@protected
614         \ekvd@assert@not@long
615         \expandafter\ekvredirectunknownNoVal\expandafter{\ekvd@set}{#1}%
616       }%
617     }%
618   }

```

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

2.2.2 Key Type Helpers

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

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

```

619 \def\ekvd@h@choice#1%
620   {%
621     \expandafter\ekvd@h@choice@
622     \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname
623     {#1}%
624   }
625 \def\ekvd@h@choice@#1#2%
626   {%
627     \ifx#1\relax
628       \ekvd@err@choice@invalid{#2}%
629       \expandafter\@gobble
630     \fi
631     #1%
632   }

```

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

2.2.3 Handling also

```

\ekvd@add@val
\ekvd@add@noval 633 \protected\long\def\ekvd@add@val#1#2#3%
\ekvd@add@aux    634   {%
\ekvd@add@aux@   635     \ekvd@assert@val{#1}%
                636     {%
                637       \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}\endcsname{{##1}}%
                638       {#1}{#2}{\ekvd@long\ekvdef}{#3}%
                639     }%
                640   }
\ekvd@add@noval 641 \protected\long\def\ekvd@add@noval#1#2#3%
\ekvd@add@aux    642   {%
\ekvd@add@aux@   643     \ekvd@assert@noval{#1}%
                644     {%
                645       \expandafter\ekvd@add@aux\csname\ekv@name\ekvd@set{#1}N\endcsname{}%
                646       {#1}{#2}\ekvdefNoVal{#3}%
                647     }%
                648   }
\ekvd@add@aux    649 \protected\long\def\ekvd@add@aux#1#2%
\ekvd@add@aux@   650   {%
\ekvd@add@aux@   651     \ekvd@extract@prefixes#1%
\ekvd@add@aux@   652     \expandafter\ekvd@add@aux@\expandafter{#1#2}%
                653   }
\ekvd@add@aux    654 \protected\long\def\ekvd@add@aux@#1#2#3#4#5%
\ekvd@add@aux@   655   {%
                656     #5%
                657     \ekvd@prot#4\ekvd@set{#2}{#1#3}%
                658   }

```

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

```

\ekvd@extract@prefixes This macro checks which prefixes were used for the definition of a macro and sets
\ekvd@extract@prefixes@ \ekvd@long and \ekvd@prot accordingly.
\ekvd@extract@prefixes@long 659 \protected\def\ekvd@extract@prefixes#1%
\ekvd@extract@prefixes@prot 660 {%
661 \expandafter\ekvd@extract@prefixes@\meaning#1\ekvd@stop
662 }

```

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.

```

663 \protected\def\ekvd@extract@prefixes@#1#2#3%
664 {%
665 \protected\def\ekvd@extract@prefixes@##1#1##2\ekvd@stop
666 {%
667 \ekvd@extract@prefixes@long
668 ##1\ekvd@mark\@firstofone#2\ekvd@mark\@gobble\ekvd@stop
669 {\let\ekvd@long\long}%
670 \ekvd@extract@prefixes@prot
671 ##1\ekvd@mark\@firstofone#3\ekvd@mark\@gobble\ekvd@stop
672 {\let\ekvd@prot\protected}%
673 }%
674 \protected\def\ekvd@extract@prefixes@long##1#2##2\ekvd@mark##3##4\ekvd@stop
675 {##3}%
676 \protected\def\ekvd@extract@prefixes@prot##1#3##2\ekvd@mark##3##4\ekvd@stop
677 {##3}%
678 }

```

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.

```

679 \begingroup
680 \edef\ekvd@tmp
681 {%
682 \endgroup
683 \ekvd@extract@prefixes@
684 {\detokenize{macro:}}%
685 {\string\long}%
686 {\string\protected}%
687 }
688 \ekvd@tmp

```

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

2.2.4 Tests

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

```

689 \protected\def\ekvd@newlet#1#2%
690 {%
691 \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\let#1#2}%
692 }
693 \protected\def\ekvd@newreg#1#2%
694 {%
695 \ifdefined#1\ekv@fi@gobble\fi\@firstofone{\csname new#2\endcsname#1}%
696 }

```

(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.

```

697 \long\def\ekvd@assert@twoargs#1%
698   {%
699     \ekvd@ifnottwoargs{#1}{\ekvd@err@missing@definition}%
700   }
701 \long\def\ekvd@ifnottwoargs#1%
702   {%
703     \ekvd@ifempty@gtwo#1\ekv@ifempty@B
704     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
705   }
706 \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.

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

```

707 \protected\def\ekvd@assert@val#1%
708   {%
709     \ekvifdefined\ekvd@set{#1}%
710     {\expandafter\ekvd@assert@val@\csname\ekv@name\ekvd@set{#1}\endcsname}%
711     {%
712       \ekvifdefinedNoVal\ekvd@set{#1}%
713       \ekvd@err@add@val@on@noval
714       {\ekvd@err@undefined@key{#1}}%
715       \@gobble
716     }%
717   }
718 \protected\def\ekvd@assert@val@#1%
719   {%
720     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop
721     \unless\ifx\ekvd@extracted@args\ekvd@one@arg@string
722     \ekvd@err@unsupported@arg
723     \fi
724     \@firstofone
725   }%
726 \protected\def\ekvd@assert@noval#1%
727   {%
728     \ekvifdefinedNoVal\ekvd@set{#1}%
729     {\expandafter\ekvd@assert@noval@\csname\ekv@name\ekvd@set{#1}N\endcsname}%
730     {%
731       \ekvifdefined\ekvd@set{#1}%
732       \ekvd@err@add@noval@on@val
733       {\ekvd@err@undefined@key{#1}}%
734       \@gobble
735     }%
736   }
737 \protected\def\ekvd@assert@noval@#1%
738   {%
739     \expandafter\ekvd@extract@args\meaning#1\ekvd@stop

```

```

740 \unless\ifx\ekvd@extracted@args\ekvd@empty
741 \ekvd@err@unsupported@arg
742 \fi
743 \@firstofone
744 }
745 \protected\def\ekvd@extract@args#1%
746 {%
747 \protected\def\ekvd@extract@args##1#1##2->##3\ekvd@stop
748 {\def\ekvd@extracted@args{##2}}}%
749 }
750 \expandafter\ekvd@extract@args\expandafter{\detokenize{macro:}}
751 \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 752 \def\ekvd@assert@arg{\ekvd@ifnoarg\ekvd@err@missing@definition}
753 \long\def\ekvd@assert@arg@msg#1%
754 {%
755 \ekvd@ifnoarg{\ekvd@err@missing@definition@msg{#1}}}%
756 }

```

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

```

\ekvd@assert@filledarg
\ekvd@ifnoarg@or@empty 757 \long\def\ekvd@assert@filledarg#1%
758 {%
759 \ekvd@ifnoarg@or@empty{#1}\ekvd@err@missing@definition
760 }
761 \long\def\ekvd@ifnoarg@or@empty#1%
762 {%
763 \ekvd@ifnoarg
764 \@firstoftwo
765 {\ekv@ifempty{#1}}}%
766 }

```

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

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

```

\ekvd@assert@not@protected@also 767 \def\ekvd@assert@not@long{\ifx\ekvd@long\long\ekvd@err@no@prefix{long}\fi}
\ekvd@assert@not@protected@also 768 \def\ekvd@assert@not@protected
\ekvd@assert@new 769 {\ifx\ekvd@prot\protected\ekvd@err@no@prefix{protected}\fi}
\ekvd@assert@not@new 770 \def\ekvd@assert@not@also{\ekvd@ifalso{\ekvd@err@no@prefix{also}}{}}
771 \def\ekvd@assert@not@long@also
772 {\ifx\ekvd@long\long\ekvd@err@no@prefix@also{long}\fi}
773 \def\ekvd@assert@not@protected@also
774 {\ifx\ekvd@prot\protected\ekvd@err@no@prefix@also{protected}\fi}
775 \def\ekvd@assert@new#1#2%
776 {\csname ekvifdefined#1\endcsname\ekvd@set{#2}{\ekvd@err@not@new}}
777 \def\ekvd@assert@not@new
778 {\ifx\ekvd@ifnew\ekvd@assert@new\ekvd@err@no@prefix{new}\fi}

```

```

779 \def\ekvd@assert@new@for@name#1%
780 {%
781   \ifx\ekvd@ifnew\ekvd@assert@new
782     \ekv@fi@firstoftwo
783   \fi
784   \@secondoftwo
785   {\ekv@ifdefined{#1}\ekvd@err@not@new}%
786   \@firstofone
787 }

```

(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.

```

788 \protected\def\ekvd@if@not@already@choice#1%
789 {%
790   \expandafter\ekvd@if@not@already@choice@a
791   \csname\ekv@name\ekvd@set{#1}\endcsname
792   {}\ekvd@h@choice\ekvd@stop
793 }
794 \protected\def\ekvd@if@not@already@choice@a
795 {%
796   \expandafter\ekvd@if@not@already@choice@b
797 }
798 \long\protected\def\ekvd@if@not@already@choice@b#1\ekvd@h@choice#2\ekvd@stop
799 {%
800   \ekv@ifempty{#2}\@firstofone\@gobble
801 }

```

(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.

```

802 \long\def\ekvd@ifspace#1%
803 {%
804   \ekvd@ifspace@#1 \ekv@ifempty@B
805   \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
806 }
807 \long\def\ekvd@ifspace@#1 % keep this space
808 {%
809   \ekv@ifempty@\ekv@ifempty@A
810 }

```

(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

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

\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
\ekvd@err@not@new

```

811 \protected\def\ekvd@errm#1{\errmessage{expkv-def Error: #1}}
812 \protected\def\ekvd@err@missing@definition

```

```

813 {\ekvd@errm{Missing definition for key '\ekvd@cur'}}
814 \protected\def\ekvd@err@missing@definition@msg#1%
815 {\ekvd@errm{Missing definition for key '\unexpanded{#1}'}}
816 \protected\def\ekvd@err@missing@type
817 {\ekvd@errm{Missing type prefix for key '\ekvd@cur'}}
818 \protected\def\ekvd@err@undefined@prefix#1%
819 {%
820 \ekvd@errm
821 {Undefined prefix '\unexpanded{#1}' found while processing '\ekvd@cur'}%
822 }
823 \protected\def\ekvd@err@undefined@key#1%
824 {%
825 \ekvd@errm
826 {Undefined key '\unexpanded{#1}' found while processing '\ekvd@cur'}%
827 }
828 \protected\def\ekvd@err@undefined@noval#1%
829 {%
830 \ekvd@errm
831 {%
832 Undefined noval key '\unexpanded{#1}' found while processing
833 '\ekvd@cur'%
834 }%
835 }
836 \protected\def\ekvd@err@no@prefix#1%
837 {\ekvd@errm{prefix '#1' not accepted in '\ekvd@cur'}}
838 \protected\def\ekvd@err@no@prefix@msg#1#2%
839 {\ekvd@errm{prefix '#2' not accepted in '\unexpanded{#1}'}}
840 \protected\def\ekvd@err@no@prefix@also#1%
841 {\ekvd@errm{'\ekvd@cur' not allowed with a '#1' key}}
842 \protected\def\ekvd@err@add@val@on@noval
843 {\ekvd@errm{'\ekvd@cur' not allowed with a NoVal key}}
844 \protected\def\ekvd@err@add@noval@on@val
845 {\ekvd@errm{'\ekvd@cur' not allowed with a value taking key}}
846 \protected\def\ekvd@err@unsupported@arg\fi\@firstofone#1%
847 {%
848 \fi
849 \ekvd@errm
850 {%
851 Existing key-macro has the unsupported argument string
852 '\ekvd@extracted@args' for key '\ekvd@cur'%
853 }%
854 }
855 \protected\def\ekvd@err@not@new
856 {\ekvd@errm{The key for '\ekvd@cur' is already defined}}
857 \protected\long\def\ekvd@err@misused@unknown
858 {\ekvd@errm{Misuse of the unknown type found while processing '\ekvd@cur'}}

```

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

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

```

859 \def\ekvd@err@choice@invalid#1%
860 {%

```

```

861     \ekvd@err@choice@invalid@#1\ekv@stop
862   }
863   \begingroup
864   \catcode40=8
865   \catcode41=8
866   \@firstofone{\endgroup
867   \def\ekvd@choice@name#1#2#3%
868     {%
869       ekvd#1(#2)#3%
870     }
871   \def\ekvd@unknown@choice@name#1#2%
872     {%
873       ekvd:u:#1(#2)%
874     }
875   \def\ekvd@err@choice@invalid@ ekvd#1(#2)#3\ekv@stop%
876     {%
877       \ekv@ifdefined{\ekvd@unknown@choice@name{#1}{#2}}%
878         {\csname\ekvd@unknown@choice@name{#1}{#2}\endcsname{#3}}%
879         {\ekvd@err{invalid choice ‘#3’ for ‘#2’ in set ‘#1’}}%
880     }
881   }

```

(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.

```

882   \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 `@`.

```

883   \catcode'\@=\ekvd@tmp

```

Index

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

A		estore	5
also	3	F	
apptoks	6	fdefault	4
B		finitial	4
bool	5	G	
boolpair	5	gapptoks	6
boolpairTF	5	gbool	5
boolTF	5	gboolpair	5
box	6	gboolpairTF	5
C		gboolTF	5
choice	7	gbox	6
code	4	gdata	5
D		gdataT	5
data	5	gdimen	6
dataT	5	gint	6
default	4	ginvbool	5
dimen	6	ginvboolTF	5
E		gskip	6
ecode	4	gstore	5
edata	5	gtoks	6
edataT	5	I	
edefault	4	initial	4
edimen	6	int	6
einitial	4	invbool	5
eint	6	invboolTF	5
\ekvchangeset	86, 89	L	
\ekvdDate	2, 5, 9, <u>15</u>	long	3
\ekvdef	274, 303, 323, 342, 362, 390, 466, 638	M	
\ekvdefinekeys	2, <u>29</u>	meta	6
\ekvdefNoVal	89, 208, 237, 646	N	
\ekvdefunknown	583	new	3
\ekvdefunknownNoVal	567	nmeta	6
\ekvdVersion	2, 5, 9, <u>15</u>	\noexpand	591, 592
\ekvverr	882	noval	4
\ekvifdefined	134, 158, 184, 709, 731	O	
\ekvifdefinedNoVal	179, 712, 728	odefault	4
\ekvlet	124, 470	oinitial	4
\ekvletNoVal	109, 145, 166	P	
\ekvparse	32, 474	protect	3
\ekvredirectunknown	602	protected	3
\ekvredirectunknownNoVal	615	Q	
\ekvset	414	qdefault	4
enoval	4		
eskip	6		

S

set	7
skip	6
smeta	6
snmeta	6
store	5

T

TeX and L^AT_EX 2_ε commands:

\ekv@exparg	108, 123, 144, 165, 194, 197, 407, 438, 460, 580, 882
\ekv@expargtwice	85, 89
\ekv@fi@firstoftwo	782
\ekv@fi@gobble	691, 695
\ekv@gobble@mark	518
\ekv@ifdefined	55, 58, 500, 555, 785, 877
\ekv@ifempty	80, 765, 800
\ekv@ifempty@	706, 809
\ekv@ifempty@A	704, 706, 805, 809
\ekv@ifempty@B	703, 704, 804, 805
\ekv@ifempty@false	704, 805
\ekv@mark	49, 52, 486, 496
\ekv@name	141, 163, 180, 187, 463, 561, 573, 596, 608, 637, 645, 710, 729, 791
\ekv@stop	49, 53, 66, 486, 498, 504, 861, 875
\ekv@strip	52, 56, 496, 508
\ekvd@add@aux	462, 633
\ekvd@add@aux@	633
\ekvd@add@noval	85, 108, 144, 165, 428, 447, 633
\ekvd@add@val	123, 271, 300, 321, 339, 360, 388, 424, 443, 633
\ekvd@carg	32, 34
\ekvd@assert@carg	103, 119, 132, 156, 532, 544, 563, 575, 598, 610, 752
\ekvd@assert@carg@msg	483, 752
\ekvd@assert@filledarg	204, 265, 294, 315, 333, 354, 382, 402, 757
\ekvd@assert@new	73, 767
\ekvd@assert@new@for@name	542, 561, 573, 596, 608, 779
\ekvd@assert@not@also	174, 547, 565, 577, 600, 612, 770
\ekvd@assert@not@also,uuuuu	767
\ekvd@assert@not@long	76, 105, 137, 161, 175, 427, 446, 451, 546, 566, 614, 767
\ekvd@assert@not@long@also	428, 447, 466
\ekvd@assert@not@new	136, 160, 173, 767

\ekvd@assert@not@protected	77, 176, 601, 613, 767
\ekvd@assert@not@protected@also	87, 767
\ekvd@assert@noval	643, 707
\ekvd@assert@noval@	707
\ekvd@assert@twoargs	248, 250, 253, 258, 434, 697
\ekvd@assert@val	456, 635, 707
\ekvd@assert@val@	707
\ekvd@choice@invalid@p	516, 525, 526, 527
\ekvd@choice@invalid@p@	518, 521
\ekvd@choice@name	210, 213, 239, 242, 453, 489, 508, 859
\ekvd@choice@p@also	526
\ekvd@choice@p@long	449
\ekvd@choice@p@long@	449
\ekvd@choice@p@new	527
\ekvd@choice@p@protect	449
\ekvd@choice@p@protected	449
\ekvd@choice@prefix	449
\ekvd@choice@prefix@	449
\ekvd@clear@prefixes	20, 46, 482
\ekvd@cur	47, 478, 483, 523, 813, 817, 821, 826, 833, 837, 841, 843, 845, 852, 856, 858
\ekvd@empty	20, 384, 740
\ekvd@err	879, 882
\ekvd@err@add@noval@on@val	732, 811
\ekvd@err@add@val@on@noval	713, 811
\ekvd@err@choice@invalid	628, 859
\ekvd@err@choice@invalid@	859
\ekvd@err@missing@definition	81, 699, 752, 759, 811
\ekvd@err@missing@definition@msg	478, 755, 811
\ekvd@err@missing@type	50, 67, 811
\ekvd@err@misused@unknown	557, 857
\ekvd@err@no@prefix	767, 769, 770, 778, 811
\ekvd@err@no@prefix@also	772, 774, 811
\ekvd@err@no@prefix@msg	523, 811
\ekvd@err@not@new	776, 785, 811
\ekvd@err@undefined@key	147, 168, 190, 714, 733, 811
\ekvd@err@undefined@noval	181, 828
\ekvd@err@undefined@prefix	60, 512, 811
\ekvd@err@unsupported@carg	722, 741, 811
\ekvd@errm	811

\ekvd@extract@args	707	\ekvd@set	31, 89, 109, 124, 134, 141, 145, 158, 163, 166, 179, 180, 184, 187, 208, 210, 213, 237, 239, 242, 274, 303, 323, 342, 362, 390, 404, 408, 439, 453, 463, 470, 489, 508, 542, 549, 561, 567, 573, 583, 596, 602, 608, 615, 637, 645, 657, 709, 710, 712, 728, 729, 731, 776, 791
\ekvd@extract@prefixes	651, 659	\ekvd@set@choice	489, 508, 535
\ekvd@extract@prefixes@	659	\ekvd@stop	661, 665, 668, 671, 674, 676, 720, 739, 747, 792, 798
\ekvd@extract@prefixes@long	659	\ekvd@t@apptoks	329
\ekvd@extract@prefixes@prot	659	\ekvd@t@bool	198
\ekvd@extracted@args	707, 852	\ekvd@t@boolpair	228
\ekvd@h@choice	453, 619, 792, 798	\ekvd@t@boolpairTF	228
\ekvd@h@choice@	619	\ekvd@t@boolTF	198
\ekvd@handle	34	\ekvd@t@box	290
\ekvd@if@not@already@choice	458, 788	\ekvd@t@choice	449
\ekvd@if@not@already@choice@a	788	\ekvd@t@code	115
\ekvd@if@not@already@choice@b	788	\ekvd@t@data	261
\ekvd@ifalso	20, 72, 83, 107, 122, 143, 164, 268, 297, 318, 336, 357, 385, 406, 437, 454, 770	\ekvd@t@dataT	261
\ekvd@ifempty@gtwo	697	\ekvd@t@default	130
\ekvd@ifnew	26, 73, 78, 101, 117, 200, 202, 230, 232, 263, 292, 313, 331, 352, 380, 400, 432, 530, 778, 781	\ekvd@t@dimen	350
\ekvd@ifnoarg	36, 41, 95, 177, 752, 763	\ekvd@t@ecode	115
\ekvd@ifnoarg@or@empty	757	\ekvd@t@edata	282
\ekvd@ifnottwoargs	697	\ekvd@t@edataT	287
\ekvd@ifspace	48, 65, 485, 503, 516, 519, 802	\ekvd@t@edefault	154
\ekvd@ifspace@	802	\ekvd@t@edimen	350
\ekvd@long	20, 69, 121, 274, 303, 323, 342, 362, 390, 422, 466, 583, 638, 669, 767, 772	\ekvd@t@einitial	171
\ekvd@mark	668, 671, 674, 676	\ekvd@t@eint	350
\ekvd@newlet	206, 234, 235, 267, 384, 689	\ekvd@t@enoval	99
\ekvd@newreg	296, 317, 335, 356, 689	\ekvd@t@eskip	350
\ekvd@noarg	32, 34	\ekvd@t@estore	396
\ekvd@one@arg@string	707	\ekvd@t@fdefault	130
\ekvd@p@also	69	\ekvd@t@finitial	171
\ekvd@p@long	69	\ekvd@t@gapptoks	329
\ekvd@p@new	69	\ekvd@t@gbool	198
\ekvd@p@protect	69	\ekvd@t@gboolpair	228
\ekvd@p@protected	69	\ekvd@t@gboolpairTF	228
\ekvd@populate@choice	449	\ekvd@t@gboolTF	198
\ekvd@populate@choice@	449	\ekvd@t@gbox	290
\ekvd@populate@choice@noarg	449	\ekvd@t@gdata	261
\ekvd@prefix	49, 52, 66	\ekvd@t@gdataT	261
\ekvd@prefix@	52	\ekvd@t@gdimen	350
\ekvd@prefix@after@p	59, 63	\ekvd@t@gint	350
\ekvd@prot	20, 70, 106, 121, 138, 162, 270, 299, 320, 338, 387, 422, 452, 506, 514, 548, 567, 583, 657, 672, 769, 774	\ekvd@t@ginvbool	198
		\ekvd@t@ginvboolTF	198
		\ekvd@t@gskip	350
		\ekvd@t@gstore	378

\ekvd@t@gtoks	<u>311</u>	\ekvd@type@default	<u>130</u>
\ekvd@t@initial	<u>171</u>	\ekvd@type@initial	
\ekvd@t@int	<u>350</u> 171, 193, 194, 195, 197	
\ekvd@t@invbool	<u>198</u>	\ekvd@type@meta	<u>398</u>
\ekvd@t@invboolTF	<u>198</u>	\ekvd@type@meta@a	<u>398</u> , <u>436</u>
\ekvd@t@meta	<u>398</u>	\ekvd@type@meta@b	<u>398</u>
\ekvd@t@nmeta	<u>398</u>	\ekvd@type@meta@c	<u>398</u>
\ekvd@t@noval	<u>99</u>	\ekvd@type@noval	<u>99</u>
\ekvd@t@odefault	<u>130</u>	\ekvd@type@reg	<u>350</u>
\ekvd@t@oinitial	<u>171</u>	\ekvd@type@set	<u>74</u>
\ekvd@t@qdefault	<u>130</u>	\ekvd@type@smeta	<u>430</u>
\ekvd@t@set	<u>74</u>	\ekvd@type@smeta@	<u>430</u>
\ekvd@t@skip	<u>350</u>	\ekvd@type@store	<u>378</u>
\ekvd@t@smeta	<u>430</u>	\ekvd@type@toks	<u>311</u>
\ekvd@t@snmeta	<u>430</u>	\ekvd@type@unknown@code	<u>553</u>
\ekvd@t@store	<u>378</u>	\ekvd@type@unknown@noval	<u>553</u>
\ekvd@t@toks	<u>311</u>	\ekvd@type@unknown@redirect	<u>589</u>
\ekvd@t@unknown	<u>553</u>	\ekvd@type@unknown@redirect-code	<u>589</u>
\ekvd@t@unknown-choice	<u>540</u>	\ekvd@type@unknown@redirect-noval	
\ekvd@t@xdata	<u>285</u> 589	
\ekvd@t@xdataT	<u>289</u>	\ekvd@unknown@choice@name	
\ekvd@t@xdimen	<u>350</u> 542, 549, 859	
\ekvd@t@xint	<u>350</u>	\ekvd@prot	<u>359</u>
\ekvd@t@xskip	<u>350</u>	toks	<u>6</u>
\ekvd@t@xstore	<u>397</u>		
\ekvd@tmp	2, 106, 108, 109,		
	121, 123, 124, 138, 144, 145, 162,		
	165, 166, 188, 193, 194, 195, 197,		
	404, 405, 407, 408, 422, 438, 439,		
	452, 465, 470, 579, 585, 680, 688, 883		
\ekvd@type@apptoks	<u>329</u>		
\ekvd@type@bool	<u>198</u>		
\ekvd@type@boolpair	<u>228</u>		
\ekvd@type@box	<u>290</u>		
\ekvd@type@choice	207, 236, <u>449</u>		
\ekvd@type@code	<u>115</u>		
\ekvd@type@data	<u>261</u>		

U	
unknown_ code	<u>7</u>
unknown_ noval	<u>7</u>
unknown_ redirect	<u>8</u>
unknown_ redirect-code	<u>7</u>
unknown_ redirect-noval	<u>8</u>
unknown-choice	<u>7</u>

X	
xdata	<u>5</u>
xdataT	<u>5</u>
xdimen	<u>6</u>
xint	<u>6</u>
xskip	<u>6</u>
xstore	<u>5</u>