

exp_k^v|DEF

a key-defining frontend for exp_k^v

Jonathan P. Spratte*

2020-07-04 v0.4

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 exp_k^v-def, this is due to CTAN-rules which don't allow | in package names since that is the pipe symbol in *nix shells.

Contents

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	19
2.2.3	Tests	20
2.2.4	Messages	21
	Index	24

*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}{\langle key \rangle = \langle value \rangle, ...}</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 p-type prefixes 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 $\langle value \rangle$.

<code>protected</code> <code>protect</code>	The following key will be defined <code>\protected</code> . Note that key-types which can't be defined expandable will always use <code>\protected</code> .
--	---

<code>long</code>	The following key will be defined <code>\long</code> .
-------------------	--

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>code</code> <code>ecode</code>	<code>code <key> = {\<definition>}</code> <code>protected long</code>
---	--

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`.

<code>noval</code> <code>enoval</code>	<code>noval <key> = {\<definition>}</code> <code>protected long</code>
---	---

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

<code>default</code> <code>qdefault</code> <code>edefault</code>	<code>default <key> = {\<definition>}</code> <code>protected long</code>
--	---

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`.

<code>initial</code> <code>oinitial</code> <code>einitial</code>	<code>initial <key> = {\<value>}</code> <code>protected long</code>
--	--

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.

<hr/> bool gbool boolTF gboolTF <hr/>	<p>bool $\langle key \rangle = \langle cs \rangle$ protected long</p> <p>The $\langle cs \rangle$ should be a single control sequence, such as <code>\iffoo</code>. This will define $\langle key \rangle$ to be a boolean key, which only takes the values <code>true</code> or <code>false</code> and will throw an error for other values. If the key is used without a $\langle value \rangle$ it'll have the same effect as if you use $\langle key \rangle = \text{true}$. <code>bool</code> and <code>gbool</code> will behave like \TeX-ifs so either be <code>\iftrue</code> or <code>\iffalse</code>. The <code>boolTF</code> and <code>gboolTF</code> variants will both take two arguments and if <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/> store estore gstore xstore <hr/>	<p>store $\langle key \rangle = \langle cs \rangle$ 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 $\langle key \rangle = \langle cs \rangle$ 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 $\langle key \rangle = \langle cs \rangle$ 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 $\{\langle value \rangle\}$.</p>
<hr/> int eint gint xint <hr/>	<p>int $\langle key \rangle = \langle cs \rangle$ protected long</p> <p>The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code>. An <code>int</code> 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 <code>eint</code> and <code>xint</code> versions will use <code>\numexpr</code> to allow basic computations in their $\langle value \rangle$. The <code>gint</code> and <code>xint</code> variants set the register globally.</p>

<u>dimen</u>	<code>dimen <key> = <cs></code>	protected long
<u>edimen</u>	The <cs> should be a single control sequence, such as \foo. This is just like int but uses a dimen register, \newdimen and \dimexpr instead.	
<u>gdimen</u>		
<u>xdimen</u>		
<u>skip</u>	<code>skip <key> = <cs></code>	protected long
<u>eskip</u>	The <cs> should be a single control sequence, such as \foo. This is just like int but uses a skip register, \newskip and \glueexpr instead.	
<u>gskip</u>		
<u>xskip</u>		
<u>toks</u>	<code>toks <key> = <cs></code>	protected long
<u>gtoks</u>	The <cs> should be a single control sequence, such as \foo. Store <value> inside of a toks-register. The g variants use \global, the app variants append <value> to the contents of that register. If <cs> is not yet defined it will be initialised with \newtoks.	
<u>apptoks</u>		
<u>gapptoks</u>		
<u>box</u>	<code>box <key> = <cs></code>	protected long
<u>gbox</u>	The <cs> should be a single control sequence, such as \foo. Typesets <value> into a \hbox and stores the result in a box register. The boxes are colour safe. expkvDEF doesn't provide a vbox type.	
<u>meta</u>	<code>meta <key> = {\<key>=<value>, ...}</code>	protected long
	This key type can set other keys, you can access the <value> which was passed to <key> inside the <key>=<value> list with #1. It works by calling a sub-\ekvset on the <key>=<value> list, so a set key will only affect that <key>=<value> list and not the current \ekvset. Since it runs in a separate \ekvset you can't use \ekvsneak using keys or similar macros in the way you normally could.	
<u>nmeta</u>	<code>nmeta <key> = {\<key>=<value>, ...}</code>	protected long
	This key type can set other keys, the difference to meta is, that this key doesn't take a value, so the <key>=<value> list is static.	
<u>smeta</u>	<code>smeta <key> = {\<set>}{\<key>=<value>, ...}</code>	protected long
	Yet another meta variant. An smeta key will take a <value> which you can access using #1, but it sets the <key>=<value> list inside of <set>, so is equal to \ekvset{\<set>}{\<key>=<value>, ...}.	
<u>snmeta</u>	<code>snmeta <key> = {\<set>}{\<key>=<value>, ...}</code>	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>.	
<u>set</u>	<code>set <key> = {\<set>}</code>	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 <code>set <key> = {\<key>}</code> . 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 T _E X error if you use an undefined <set>.	

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

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>	<code>protected long</code>
-----------------------	--	-----------------------------

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}
    ,data
}

```

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

```

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

```

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

Jonathan P. Spratte

Age *young*

Hobbies \TeX coding

Some User

Age *unknown*

Hobbies Reading Documentation

Anybody

Age *any*

1.5 License

Copyright © 2020 Jonathan P. Spratte

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

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

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

2 Implementation

2.1 The L^AT_EX Package

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

```

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

2.2 The Generic Code

The rest of this implementation will be the generic code.

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

```

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

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

```

\ekvdDate
15 \def\ekvdVersion{0.4}
16 \def\ekvdDate{2020-07-04}
```

(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 `expkVIDEF` was initialized.

```

\ekvd@long expkVIDEF will use \ekvd@long and \ekvd@prot to store whether a key should be defined
\ekvd@prot as \long or \protected, and we have to clear them for every new key. By default they'll
\ekvd@clear@prefixes just be empty.
\ekvd@empty
20 \def\ekvd@empty{}
21 \protected\def\ekvd@clear@prefixes
22   {%
```

```

23 \let\ekvd@long\ekvd@empty
24 \let\ekvd@prot\ekvd@empty
25 }
26 \ekvd@clear@prefixes

```

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

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

```

27 \protected\def\ekvdefinekeys#1%
28 {%
29 \def\ekvd@set{#1}%
30 \ekvparse\ekvd@noarg\ekvd@
31 }

```

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

\ekvd@noarg \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.

\ekvd@

```

32 \protected\def\ekvd@noarg#1{\ekvd@{#1}\ekvd@noarg@mark}
33 \protected\long\def\ekvd@#1#2%
34 {%
35 \ekvd@clear@prefixes
36 \ekvd@ifspace{#1}%
37 {\ekvd@prefix\ekv@mark#1\ekv@stop{#2}}%
38 {\ekvd@err@missing@prefix{#1}}%
39 }

```

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

\ekvd@prefix **\ekvd@prefix@** **expkvDEF** 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).

```

40 \protected\def\ekvd@prefix#1 {\ekv@strip{#1}\ekvd@prefix@\ekv@mark}
41 \protected\def\ekvd@prefix@#1#2\ekv@stop
42 {%
43 \ekv@ifdefined{ekvd@t@#1}%
44 {\ekv@strip{#2}{\csname ekvd@t@#1\endcsname}}%
45 {%
46 \ekv@ifdefined{ekvd@p@#1}%
47 {\csname ekvd@p@#1\endcsname{#2}}%
48 {\ekvd@err@undefined@prefix{#1}\@gobble}%
49 }%
50 }

```

(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.

```

51 \protected\def\ekvd@prefix@after@p#1%
52 {%
53   \ekvd@ifspace{#1}%
54   {\ekvd@prefix#1\ekv@stop}%
55   {%
56     \expandafter\ekvd@err@missing@prefix\expandafter{\ekv@gobble@mark#1}%
57     \@gobble
58   }%
59 }

```

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

`\ekvd@p@long` Define the `@p@` type prefixes, they all just store some information in a temporary macro
`\ekvd@p@protected` and call `\ekvd@prefix@after@p`.
`\ekvd@p@protect`

```

60 \protected\def\ekvd@p@long{\let\ekvd@long\long\ekvd@prefix@after@p}
61 \protected\def\ekvd@p@protected{\let\ekvd@prot\protected\ekvd@prefix@after@p}
62 \let\ekvd@p@protect\ekvd@p@protected

```

(End definition for `\ekvd@p@long`, `\ekvd@p@protected`, and `\ekvd@p@protect`.)

2.2.1 Key Types

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

```

63 \protected\def\ekvd@t@set#1#2%
64 {%
65   \ekvd@assert@not@long{set #1}%
66   \ekvd@assert@not@protected{set #1}%
67   \ekvd@ifnoarg{#2}%
68   {\ekvdefNoVal\ekvd@set{#1}{\ekvchangeset{#1}}}%
69   {%
70     \ekv@ifempty{#2}%
71     {\ekvd@err@missing@definition{set #1}}%
72     {\ekvdefNoVal\ekvd@set{#1}{\ekvchangeset{#2}}}%
73   }%
74 }

```

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

```

75 \protected\long\def\ekvd@type@noval#1#2#3#4%
76 {%
77   \ekvd@assert@arg{#1noval #3}{#4}%
78   {%
79     \ekvd@assert@not@long{#1noval #3}%
80     \ekvd@prot#2\ekvd@tmp{#4}%
81     \ekvletNoVal\ekvd@set{#3}\ekvd@tmp
82   }%

```

```

83 }
84 \protected\def\ekvd@t@noval{\ekvd@type@noval{}\def}
85 \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 `explkv` doesn't provide an `\ekvedef`.

```

\ekvd@t@code
\ekvd@t@ecode
86 \protected\long\def\ekvd@type@code#1#2#3#4%
87 {%
88   \ekvd@assert@arg{#1code #3}{#4}
89   {%
90     \ekvd@prot\ekvd@long#2\ekvd@tmp##1{#4}%
91     \ekvlet\ekvd@set{#3}\ekvd@tmp
92   }%
93 }
94 \protected\def\ekvd@t@code{\ekvd@type@code{}\def}
95 \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@t@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.

```

96 \protected\long\def\ekvd@type@default#1#2#3#4%
97 {%
98   \ekvd@assert@arg{#1default #3}{#4}%
99   {%
100     \ekvifdefined\ekvd@set{#3}%
101     {%
102       \ekvd@assert@not@long{#1default #3}%
103       \ekvd@prot\edef\ekvd@tmp
104       {%
105         \unexpanded\expandafter#2%
106         {\csname\ekv@name\ekvd@set{#3}\endcsname{#4}}%
107       }%
108       \ekvletNoVal\ekvd@set{#3}\ekvd@tmp
109     }%
110     {\ekvd@err@undefined@key{#3}}%
111   }%
112 }
113 \protected\def\ekvd@t@default{\ekvd@type@default{}\{}}
114 \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`.

```

115 \protected\long\def\ekvd@t@edefault#1#2%
116 {%
117   \ekvd@assert@arg{edefault #1}{#2}%
118   {%

```

```

119     \ekvifdefined\ekvd@set{#1}%
120     {%
121         \ekvd@assert@not@long{edefault #1}%
122         \ekvd@prot\edef\ekvd@tmp
123             {\csname\ekv@name\ekvd@set{#1}\endcsname{#2}}%
124         \ekvletNoVal\ekvd@set{#1}\ekvd@tmp
125     }%
126     {\ekvd@err@undefined@key{#1}}%
127 }%
128 }

```

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

```

\ekvd@t@initial
\ekvd@t@oinitial 129 \long\def\ekvd@t@initial#1#2%
\ekvd@t@einitial 130 {%
131     \ekvd@assert@arg{initial #1}{#2}%
132     {%
133         \ekvifdefined\ekvd@set{#1}%
134         {%
135             \ekvd@assert@not@long{initial #1}%
136             \ekvd@assert@not@protected{initial #1}%
137             \csname\ekv@name\ekvd@set{#1}\endcsname{#2}%
138         }%
139         {\ekvd@err@undefined@key{#1}}%
140     }%
141 }
142 \long\def\ekvd@t@oinitial#1#2%
143 {%
144     \ekvd@assert@arg{oinitial #1}{#2}%
145     {%
146         \ekvifdefined\ekvd@set{#1}%
147         {%
148             \ekvd@assert@not@long{oinitial #1}%
149             \ekvd@assert@not@protected{oinitial #1}%
150             \csname\ekv@name\ekvd@set{#1}\expandafter\endcsname\expandafter{#2}%
151         }%
152         {\ekvd@err@undefined@key{#1}}%
153     }%
154 }
155 \long\def\ekvd@t@einitial#1#2%
156 {%
157     \ekvd@assert@arg{einitial #1}{#2}%
158     {%
159         \ekvifdefined\ekvd@set{#1}%
160         {%
161             \ekvd@assert@not@long{einitial #1}%
162             \ekvd@assert@not@protected{einitial #1}%
163             \edef\ekvd@tmp{#2}%
164             \csname\ekv@name\ekvd@set{#1}\expandafter\endcsname\expandafter
165                 {\ekvd@tmp}%
166         }%
167         {\ekvd@err@undefined@key{#1}}%
168     }%
169 }

```

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

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 $\langle \text{key} \rangle = \text{true}$. The true and false actions are always just \letting the macro in #7 to some other macro (e.g., \iftrue).

```

\ekvd@type@bool
\ekvd@t@bool
\ekvd@t@gbool
\ekvd@t@boolTF
\ekvd@t@gboolTF
170 \protected\def\ekvd@type@bool#1#2#3#4#5#6#7%
171   {%
172     \ekvd@assert@filledarg{#1bool#2 #6}{#7}%
173     {%
174       \ekvd@newlet#7#5%
175       \ekvd@type@choice{#1bool#2}{#6}%
176       \protected\ekvdefNoVal\ekvd@set{#6}{#3\let#7#4}%
177       \protected\expandafter\def
178         \csname\ekvd@choice@name\ekvd@set{#6}{true}\endcsname
179         {#3\let#7#4}%
180       \protected\expandafter\def
181         \csname\ekvd@choice@name\ekvd@set{#6}{false}\endcsname
182         {#3\let#7#5}%
183     }%
184   }
185 \protected\def\ekvd@t@bool{\ekvd@type@bool{}}{\iftrue\iffalse}
186 \protected\def\ekvd@t@gbool{\ekvd@type@bool g}{\global\iftrue\iffalse}
187 \protected\def\ekvd@t@boolTF{\ekvd@type@bool{}}{TF}{\@firstoftwo\@secondoftwo}
188 \protected\def\ekvd@t@gboolTF
189   {\ekvd@type@bool g}{TF}{\global\@firstoftwo\@secondoftwo}

```

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

```

\ekvd@type@data
\ekvd@t@data
\ekvd@t@gdata
\ekvd@t@dataT
\ekvd@t@gdataT
190 \protected\def\ekvd@type@data#1#2#3#4#5#6#7%
191   {%
192     \ekvd@assert@filledarg{#1data#2 #6}{#7}%
193     {%
194       \ekvd@newlet#7#3%
195       \protected\ekvd@long\ekvdef\ekvd@set{#6}{\long#4#7####1#5{####1{##1}}}%
196     }%
197   }
198 \protected\def\ekvd@t@data{\ekvd@type@data{}}{\@secondoftwo\def{####2}}
199 \protected\def\ekvd@t@edata{\ekvd@type@data e}{\@secondoftwo\edef{####2}}
200 \protected\def\ekvd@t@gdata{\ekvd@type@data g}{\@secondoftwo\gdef{####2}}
201 \protected\def\ekvd@t@xdata{\ekvd@type@data x}{\@secondoftwo\xdef{####2}}
202 \protected\def\ekvd@t@dataT{\ekvd@type@data{T}\@gobble\def{}}
203 \protected\def\ekvd@t@edataT{\ekvd@type@data eT}\@gobble\edef{}}
204 \protected\def\ekvd@t@gdataT{\ekvd@type@data gT}\@gobble\gdef{}}
205 \protected\def\ekvd@t@xdataT{\ekvd@type@data xT}\@gobble\xdef{}}

```

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

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.

```

206 \protected\def\ekvd@type@box#1#2#3#4%
207   {%

```

```

208 \ekvd@assert@filledarg{#1box #3}{#4}%
209 {%
210 \ekvd@newreg#4{box}%
211 \protected\ekvd@long\ekvdef\ekvd@set{#3}%
212 {#2\setbox#4\hbox{\begingroup##1\endgroup}}}%
213 }%
214 }
215 \protected\def\ekvd@t@box{\ekvd@type@box{}}{}
216 \protected\def\ekvd@t@gbox{\ekvd@type@box g\global}

```

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

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

```

\ekvd@t@toks 217 \protected\def\ekvd@type@toks#1#2#3#4%
\ekvd@t@gtoks 218 {%
219 \ekvd@assert@filledarg{#1toks #3}{#4}%
220 {%
221 \ekvd@newreg#4{toks}%
222 \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{##1}}%
223 }%
224 }
225 \protected\def\ekvd@t@toks{\ekvd@type@toks{}}{}
226 \protected\def\ekvd@t@gtoks{\ekvd@type@toks{g}\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 227 \protected\def\ekvd@type@apptoks#1#2#3#4%
\ekvd@t@gapptoks 228 {%
229 \ekvd@assert@filledarg{#1apptoks #3}{#4}%
230 {%
231 \ekvd@newreg#4{toks}%
232 \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4\expandafter{\the#4##1}}%
233 }%
234 }
235 \protected\def\ekvd@t@apptoks{\ekvd@type@apptoks{}}{}
236 \protected\def\ekvd@t@gapptoks{\ekvd@type@apptoks{g}\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 $\langle register \rangle = \langle value \rangle$.

```

\ekvd@t@int 237 \protected\def\ekvd@type@reg#1#2#3#4#5#6#7%
\ekvd@t@eint 238 {%
\ekvd@t@gint 239 \ekvd@assert@filledarg{#1 #6}{#7}%
\ekvd@t@xint 240 {%
\ekvd@t@dimen 241 \ekvd@newreg#7{#2}%
\ekvd@t@edimen 242 \protected\ekvd@long\ekvdef\ekvd@set{#6}{#3#7=#4##1#5\relax}%
\ekvd@t@gdimen 243 }%
\ekvd@t@xdimen 244 }
\ekvd@t@skip 245 \protected\def\ekvd@t@int{\ekvd@type@reg{int}{count}{}}{}
\ekvd@t@eskip 246 \protected\def\ekvd@t@eint{\ekvd@type@reg{eint}{count}{}\numexpr\relax}
\ekvd@t@gskip 247 \protected\def\ekvd@t@gint{\ekvd@type@reg{gint}{count}{\global{}}{}
\ekvd@t@xskip 248 \protected\def\ekvd@t@xint{\ekvd@type@reg{xint}{count}{\global\numexpr\relax}

```

```

249 \protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{dimen}{}}{}{}
250 \protected\def\ekvd@t@edimen{\ekvd@type@reg{edimen}{dimen}{}}\dimexpr\relax
251 \protected\def\ekvd@t@gdimen{\ekvd@type@reg{gdimen}{dimen}\global{}}{}
252 \protected\def\ekvd@t@xdimen{\ekvd@type@reg{xdimen}{dimen}\global\dimexpr\relax}
253 \protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{skip}}{}{}{}
254 \protected\def\ekvd@t@eskip{\ekvd@type@reg{eskip}{skip}}\glueexpr\relax
255 \protected\def\ekvd@t@gskip{\ekvd@type@reg{gskip}{skip}\global{}}{}
256 \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.

```

\ekvd@t@store
\ekvd@t@gstore
257 \protected\def\ekvd@type@store#1#2#3#4%
258   {%
259     \ekvd@assert@filledarg{#1store #3}{#4}%
260     {%
261       \unless\ifdefined#4\let#4\ekvd@empty\fi
262       \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{\unexpanded{##1}}}%
263     }%
264   }
265 \protected\def\ekvd@t@store{\ekvd@type@store{}}\edef{}
266 \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.

```

\ekvd@t@estore
\ekvd@t@xstore
267 \protected\def\ekvd@type@estore#1#2#3#4%
268   {%
269     \ekvd@assert@filledarg{#1store #3}{#4}%
270     {%
271       \ekvd@newlet#4\ekvd@empty
272       \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{##1}}%
273     }%
274   }
275 \protected\def\ekvd@t@estore{\ekvd@type@estore{e}}\edef{}
276 \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 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
277 \protected\long\def\ekvd@type@meta#1#2#3#4#5%
278   {%
279     \ekvd@assert@filledarg{#1meta #4}{#5}%
280     {%
281       \edef\ekvd@tmp{\ekvd@set}%
282       \expandafter\ekvd@type@meta@a\expandafter{\ekvd@tmp}{#5}{#3}%
283       #2\ekvd@set{#4}\ekvd@tmp
284     }%
285   }
286 \protected\long\def\ekvd@type@meta@a#1#2%
287   {%
288     \expandafter\ekvd@type@meta@b\expandafter{\ekvset{#1}{#2}}%
289   }

```

```

290 \protected\def\ekvd@type@meta@b
291   {%
292     \expandafter\ekvd@type@meta@c\expandafter
293   }
294 \protected\long\def\ekvd@type@meta@c#1#2%
295   {%
296     \ekvd@prot\ekvd@long\def\ekvd@tmp#2{#1}%
297   }
298 \protected\def\ekvd@t@meta{\ekvd@type@meta{}}\ekvlet{##1}}
299 \protected\long\def\ekvd@t@nmeta#1#2%
300   {%
301     \ekvd@assert@not@long{nmeta #1}%
302     \ekvd@type@meta n\ekvletNoVal{#1}{#2}%
303   }

```

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

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

```

\ekvd@type@smeta@
\ekvd@t@smeta
\ekvd@t@snmeta
304 \protected\long\def\ekvd@type@smeta#1#2#3#4#5%
305   {%
306     \ekvd@assert@twoargs{s#1meta #4}{#5}%
307     {%
308       \ekvd@type@meta@a#5{#3}%
309       #2\ekvd@set{#4}\ekvd@tmp
310     }%
311   }
312 \protected\def\ekvd@t@smeta{\ekvd@type@smeta{}}\ekvlet{##1}}
313 \protected\long\def\ekvd@t@snmeta#1#2%
314   {%
315     \ekvd@assert@not@long{snmeta #1}%
316     \ekvd@type@smeta n\ekvletNoVal{#1}{#2}%
317   }

```

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

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

```

\ekvd@type@choice
\ekvd@populate@choice
\ekvd@populate@choice@
\ekvd@populate@choice@noarg
\ekvd@choice@prefix
\ekvd@choice@prefix@
\ekvd@choice@p@protected
\ekvd@choice@p@protect
\ekvd@choice@p@long
\ekvd@choice@p@long@
\ekvd@t@choice
318 \protected\def\ekvd@type@choice#1#2%
319   {%
320     \ekvd@assert@not@long{#1 #2}%
321     \ekvd@prot\edef\ekvd@tmp##1%
322     {%
323       \unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#2}{##1}}%
324     }%
325     \ekvlet\ekvd@set{#2}\ekvd@tmp
326   }

```

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

```

327 \protected\def\ekvd@populate@choice

```

```

328 {%
329     \ekvparsed\ekvd@populate@choice@noarg\ekvd@populate@choice@
330 }
331 \protected\long\def\ekvd@populate@choice@noarg#1%
332 {%
333     \expandafter\ekvd@err@missing@definition\expandafter{\ekvd@set@choice : #1}%
334 }
\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.
335 \protected\long\def\ekvd@populate@choice@#1#2%
336 {%
337     \ekvd@clear@prefixes
338     \expandafter\ekvd@assert@arg\expandafter{\ekvd@set@choice : #1}{#2}%
339     {%
340         \ekvd@ifspace{#1}%
341         {\ekvd@choice@prefix\ekv@mark#1\ekv@stop}%
342         {%
343             \expandafter\def
344             \csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname
345             }%
346         {#2}%
347     }%
348 }
349 \protected\def\ekvd@choice@prefix#1
350 {%
351     \ekv@strip{#1}\ekvd@choice@prefix@\ekv@mark
352 }
353 \protected\def\ekvd@choice@prefix@#1#2\ekv@stop
354 {%
355     \ekv@ifdefined{\ekvd@choice@p@#1}%
356     {%
357         \csname \ekvd@choice@p@#1\endcsname
358         \ekvd@ifspace{#2}%
359         {\ekvd@choice@prefix#2\ekv@stop}%
360         {%
361             \ekvd@prot\expandafter\def
362             \csname
363                 \ekv@strip{#2}{\ekvd@choice@name\ekvd@set\ekvd@set@choice}%
364             \endcsname
365             }%
366         }%
367     {\ekvd@err@undefined@prefix{#1}\@gobble}%
368 }
369 \protected\def\ekvd@choice@p@protected{\let\ekvd@prot\protected}
370 \let\ekvd@choice@p@protect\ekvd@choice@p@protected
371 \protected\def\ekvd@choice@p@long\ekvd@ifspace#1%
372 {%
373     \expandafter\ekvd@choice@p@long\expandafter{\ekv@gobble@mark#1}%
374     \ekvd@ifspace{#1}%
375 }
376 \protected\def\ekvd@choice@p@long@#1%

```

```

377     {%
378       \expandafter\ekvd@err@no@long\expandafter
379       {\ekvd@set@choice : long #1}%
380     }

```

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.

```

381 \protected\long\def\ekvd@t@choice#1#2%
382 {%
383   \ekvd@assert@arg{choice #1}{#2}%
384   {%
385     \ekvd@type@choice{choice}{#1}%
386     \def\ekvd@set@choice{#1}%
387     \ekvd@populate@choice{#2}%
388   }%
389 }

```

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

\ekvd@t@unknown-choice

```

390 \protected\long\expandafter\def\csname ekvd@t@unknown-choice\endcsname#1#2%
391 {%
392   \ekvd@assert@arg{unknown-choice #1}{#2}%
393   {%
394     \ekvd@assert@not@long{unknown-choice #1}%
395     \ekvd@prot\expandafter
396     \def\csname\ekvd@unknown@choice@name\ekvd@set{#1}\endcsname##1{#2}%
397   }%
398 }

```

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

2.2.2 Key Type Helpers

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

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

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

```

399 \def\ekvd@h@choice#1%
400 {%
401   \expandafter\ekvd@h@choice@
402   \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname
403   {#1}%
404 }
405 \def\ekvd@h@choice@#1#2%
406 {%
407   \ifx#1\relax
408     \ekvd@err@choice@invalid{#2}%
409     \expandafter\@gobble
410   \fi
411   #1%
412 }

```

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

2.2.3 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>`.

```

413 \protected\def\ekvd@newlet#1#2%
414   {%
415     \unless\ifdefined#1\let#1#2\fi
416   }
417 \protected\def\ekvd@newreg#1#2%
418   {%
419     \unless\ifdefined#1\csname new#2\endcsname#1\fi
420   }

```

(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.

```

421 \long\def\ekvd@assert@twoargs#1#2%
422   {%
423     \ekvd@ifnottwoargs{#2}%
424     {\ekvd@err@missing@definition{#1}}%
425   }
426 \long\def\ekvd@ifnottwoargs#1%
427   {%
428     \ekvd@ifempty@gtwo#1\ekv@ifempty@B
429     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
430   }
431 \long\def\ekvd@ifempty@gtwo#1#2{\ekv@ifempty@A\ekv@ifempty@B}

```

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

`\ekvd@assert@arg` An `\ifx` comparison would fail if the first token of the user input is a TeX-if primitive or the user input contains unbalanced TeX-if constructs, so the test is done using TeX's argument grabbing logic.

```

432 \long\def\ekvd@assert@arg#1#2%
433   {%
434     \ekvd@ifnoarg{#2}%
435     {\ekvd@err@missing@definition{#1}}%
436   }
437 \long\def\ekvd@ifnoarg#1%
438   {%
439     \ekvd@ifnoarg@\ekvd@ifnoarg@mark#1\ekvd@ifnoarg@mark\ekvd@ifnoarg@t
440     \ekvd@ifnoarg@mark\ekvd@ifnoarg@mark\ekvd@ifnoarg@mark\@secondoftwo
441   }
442 \long\def\ekvd@ifnoarg@#1\ekvd@ifnoarg@mark\ekvd@ifnoarg@mark\ekvd@ifnoarg@mark{}
443 \long\def\ekvd@ifnoarg@t
444   \ekvd@ifnoarg@mark\ekvd@ifnoarg@mark\ekvd@ifnoarg@mark\@secondoftwo#1#2%
445   {%
446     #1%
447   }

```

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

```

\ekvd@assert@filledarg
\ekvd@ifnoarg@or@empty
448 \long\def\ekvd@assert@filledarg#1#2%
449 {%
450     \ekvd@ifnoarg@or@empty{#2}%
451     {\ekvd@err@missing@definition{#1}}%
452 }
453 \long\def\ekvd@ifnoarg@or@empty#1%
454 {%
455     \ekvd@ifnoarg{#1}%
456     \@firstoftwo
457     {\ekv@ifempty{#1}}%
458 }

```

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

```

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

```

Some key-types don't want to be \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.

```

459 \long\def\ekvd@assert@not@long#1%
460 {%
461     \ifx\ekvd@long\long\ekvd@err@no@long{#1}\fi
462 }
463 \long\def\ekvd@assert@not@protected#1%
464 {%
465     \ifx\ekvd@prot\protected\ekvd@err@no@protected{#1}\fi
466 }

```

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

```

\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.

```

467 \long\def\ekvd@ifspace#1%
468 {%
469     \ekvd@ifspace@#1 \ekv@ifempty@B
470     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
471 }
472 \long\def\ekvd@ifspace@#1 % keep this space
473 {%
474     \ekv@ifempty@\ekv@ifempty@A
475 }

```

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

2.2.4 Messages

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

```

\ekvd@err@missing@definition
\ekvd@err@missing@prefix
\ekvd@err@undefined@prefix
\ekvd@err@undefined@key
\ekvd@err@no@protected
\ekvd@err@no@long

```

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

```

476 \protected\def\ekvd@err@missing@definition#1%
477     {\errmessage{expkv-def Error: Missing definition for key '\unexpanded{#1}'} }
478 \protected\def\ekvd@err@missing@prefix#1%
479     {\errmessage{expkv-def Error: Missing prefix for key '\unexpanded{#1}'} }
480 \protected\def\ekvd@err@undefined@prefix#1%

```

```

481 {\errmessage{expkv-def Error: Undefined prefix '\unexpanded{#1}'}}
482 \protected\def\ekvd@err@undefined@key#1%
483 {\errmessage{expkv-def Error: Undefined key '\unexpanded{#1}'}}
484 \protected\def\ekvd@err@no@protected#1%
485 {%
486   \errmessage
487     {expkv-def Error: prefix 'protected' not accepted for '\unexpanded{#1}'}%
488 }
489 \protected\def\ekvd@err@no@long#1%
490 {%
491   \errmessage
492     {expkv-def Error: prefix 'long' not accepted for '\unexpanded{#1}'}%
493 }

```

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

\ekvd@err@choice@invalid The expandable error messages use \ekvd@err, which is just like \ekv@err from `expl3` 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.

```

494 \def\ekvd@err@choice@invalid#1%
495 {%
496   \ekvd@err@choice@invalid@#1\ekv@stop
497 }
498 \begingroup
499 \catcode40=8
500 \catcode41=8
501 \@firstofone{\endgroup
502 \def\ekvd@choice@name#1#2#3%
503 {%
504   ekvd#1(#2)#3%
505 }
506 \def\ekvd@unknown@choice@name#1#2%
507 {%
508   ekvd:u:#1(#2)%
509 }
510 \def\ekvd@err@choice@invalid@ ekvd#1(#2)#3\ekv@stop%
511 {%
512   \ekv@ifdefined{\ekvd@unknown@choice@name{#1}{#2}}%
513     {\csname\ekvd@unknown@choice@name{#1}{#2}\endcsname{#3}}%
514     {\ekvd@err{invalid choice '#3' ('#2', set '#1')}}%
515 }
516 }
517 \begingroup
518 \edef\ekvd@err
519 {%
520   \endgroup
521   \unexpanded{\long\def\ekvd@err}##1%
522   {%
523     \unexpanded{\expandafter\ekv@err@\@firstofone}%
524     {\unexpanded{\expandafter{\csname ! expkv-def Error:\endcsname}##1.}%
525     \unexpanded{\ekv@stop}}%

```

```

526         }%
527     }
528     \ekvd@err
    (End definition for \ekvd@err@choice@invalid and others.)
    Now everything that's left is to reset the category code of @.
529     \catcode'\@=\ekvd@tmp

```

Index

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

A		<code>\gdef</code> 200, 204
<code>apptoks</code>	5	<code>gdimen</code> 5
B		<code>gint</code> 4
<code>bool</code>	4	<code>gskip</code> 5
<code>boolTF</code>	4	<code>gstore</code> 4
<code>box</code>	5	<code>gtoks</code> 5
C		I
<code>choice</code>	6	<code>initial</code> 3
<code>code</code>	3	<code>int</code> 4
D		L
<code>data</code>	4	<code>long</code> 3
<code>dataT</code>	4	M
<code>default</code>	3	<code>meta</code> 5
<code>dimen</code>	5	N
E		<code>nmeta</code> 5
<code>ecode</code>	3	<code>noval</code> 3
<code>edata</code>	4	O
<code>edataT</code>	4	<code>oinitial</code> 3
<code>edefault</code>	3	P
<code>edimen</code>	5	<code>protect</code> 3
<code>einitial</code>	3	<code>protected</code> 3
<code>eint</code>	4	Q
<code>\ekvchangeset</code>	68, 72	<code>qdefault</code> 3
<code>\ekvdDate</code>	2, 5, 9, 15	S
<code>\ekvdef</code> ...	195, 211, 222, 232, 242, 262, 272	<code>set</code> 5
<code>\ekvdefinekeys</code>	2, 27	<code>skip</code> 5
<code>\ekvdefNoVal</code>	68, 72, 176	<code>smeta</code> 5
<code>\ekvdVersion</code>	2, 5, 9, 15	<code>snmeta</code> 5
<code>\ekvifdefined</code>	100, 119, 133, 146, 159	<code>store</code> 4
<code>\ekvlet</code>	91, 298, 312, 325	T
<code>\ekvletNoVal</code>	81, 108, 124, 302, 316	<code>T_EX and L^AT_EX_{2_ε} commands:</code>
<code>\ekvparse</code>	30, 329	<code>\ekv@err@</code> 523
<code>\ekvset</code>	288	<code>\ekv@gobble@mark</code> 56, 373
<code>enoval</code>	3	<code>\ekv@ifdefined</code> 43, 46, 355, 512
<code>eskip</code>	5	<code>\ekv@ifempty</code> 70, 457
<code>estore</code>	4	<code>\ekv@ifempty@</code> 431, 474
G		<code>\ekv@ifempty@A</code> 429, 431, 470, 474
<code>gapptoks</code>	5	<code>\ekv@ifempty@B</code> 428, 429, 469, 470
<code>gbool</code>	4	<code>\ekv@ifempty@false</code> 429, 470
<code>gboolTF</code>	4	<code>\ekv@mark</code> 37, 40, 341, 351
<code>gbox</code>	5	
<code>gdata</code>	4	
<code>gdataT</code>	4	

\ekv@name	106, 123, 137, 150, 164
\ekv@stop	37, 41, 54, 341, 353, 359, 496, 510, 525
\ekv@strip	40, 44, 351, 363
\ekvd@	30, 32
\ekvd@assert@arg	77, 88, 98, 117, 131, 144, 157, 338, 383, 392, 432
\ekvd@assert@filledarg	172, 192, 208, 219, 229, 239, 259, 269, 279, 448
\ekvd@assert@not@long	65, 79, 102, 121, 135, 148, 161, 301, 315, 320, 394, 459
\ekvd@assert@not@protected	66, 136, 149, 162, 459
\ekvd@assert@twoargs	306, 421
\ekvd@choice@name	178, 181, 323, 344, 363, 494
\ekvd@choice@p@long	318
\ekvd@choice@p@long@	318
\ekvd@choice@p@protect	318
\ekvd@choice@p@protected	318
\ekvd@choice@prefix	318
\ekvd@choice@prefix@	318
\ekvd@clear@prefixes	20, 35, 337
\ekvd@empty	20, 261, 271
\ekvd@err	494
\ekvd@err@choice@invalid	408, 494
\ekvd@err@choice@invalid@	494
\ekvd@err@missing@definition	71, 333, 424, 435, 451, 476
\ekvd@err@missing@prefix	38, 56, 476
\ekvd@err@no@long	378, 461, 476
\ekvd@err@no@protected	465, 476
\ekvd@err@undefined@key	110, 126, 139, 152, 167, 476
\ekvd@err@undefined@prefix	48, 367, 476
\ekvd@h@choice	323, 399
\ekvd@h@choice@	399
\ekvd@ifempty@gtwo	421
\ekvd@ifnoarg	67, 432, 455
\ekvd@ifnoarg@	432
\ekvd@ifnoarg@mark	439, 440, 442, 444
\ekvd@ifnoarg@or@empty	448
\ekvd@ifnoarg@t	439, 443
\ekvd@ifnottwoargs	421
\ekvd@ifspace	36, 53, 340, 358, 371, 374, 467
\ekvd@ifspace@	467
\ekvd@long	20, 60, 90, 195, 211, 222, 232, 242, 262, 272, 296, 461
\ekvd@newlet	174, 194, 271, 413
\ekvd@newreg	210, 221, 231, 241, 413
\ekvd@noarg	30, 32
\ekvd@noarg@mark	32, 440, 442, 444
\ekvd@p@long	60
\ekvd@p@protect	60
\ekvd@p@protected	60
\ekvd@populate@choice	318
\ekvd@populate@choice@	318
\ekvd@populate@choice@noarg	318
\ekvd@prefix	37, 40, 54
\ekvd@prefix@	40
\ekvd@prefix@after@p	51, 60, 61
\ekvd@prot	20, 61, 80, 90, 103, 122, 296, 321, 361, 369, 395, 465
\ekvd@set	29, 68, 72, 81, 91, 100, 106, 108, 119, 123, 124, 133, 137, 146, 150, 159, 164, 176, 178, 181, 195, 211, 222, 232, 242, 262, 272, 281, 283, 309, 323, 325, 344, 363, 396
\ekvd@set@choice	333, 338, 344, 363, 379, 386
\ekvd@t@apptoks	227
\ekvd@t@bool	170
\ekvd@t@boolTF	170
\ekvd@t@box	206
\ekvd@t@choice	318
\ekvd@t@code	86
\ekvd@t@data	190
\ekvd@t@dataT	190
\ekvd@t@default	96
\ekvd@t@dimen	237
\ekvd@t@ecode	86
\ekvd@t@edata	199
\ekvd@t@edataT	203
\ekvd@t@edefault	115
\ekvd@t@edimen	237
\ekvd@t@einitial	129
\ekvd@t@eint	237
\ekvd@t@enoval	75
\ekvd@t@eskip	237
\ekvd@t@estore	267
\ekvd@t@gapptoks	227
\ekvd@t@gbool	170
\ekvd@t@gboolTF	170
\ekvd@t@gbox	206
\ekvd@t@gdata	190

