

exp_k^v|DEF

a key-defining frontend for exp_k^v

Jonathan P. Spratte*

2020-02-27 v0.1

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	5
1.4	Example	5
1.5	License	6
2	Implementation	7
2.1	The L ^A T _E X Package	7
2.2	The Generic Code	7
2.2.1	Key Types	9
2.2.2	Key Type Helpers	16
2.2.3	Tests	16
2.2.4	Messages	18

Index	21
--------------	-----------

*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> <div style="text-align: right;"><code>protected long</code></div> <p>Define <code><key></code> to expand to <code><definition></code>. The <code><key></code> will require a <code><value></code> for which you can use <code>#1</code> inside <code><definition></code>. The <code>ecode</code> variant will fully expand <code><definition></code> inside an <code>\edef</code>.</p>
<code>noval</code> <code>enoval</code>	<code>noval <key> = {<definition>}</code> <div style="text-align: right;"><code>protected long</code></div> <p>The <code>noval</code> type defines <code><key></code> to expand to <code><definition></code>. The <code><key></code> will not take a <code><value></code>. <code>enoval</code> fully expands <code><definition></code> inside an <code>\edef</code>.</p>
<code>default</code> <code>qdefault</code> <code>edefault</code>	<code>default <key> = {<definition>}</code> <div style="text-align: right;"><code>protected long</code></div> <p>This serves to place a default <code><value></code> for a <code><key></code> that takes an argument, the <code><key></code> can be of any argument-grabbing kind, and when used without a <code><value></code> it will be passed <code><definition></code> instead. The <code>qdefault</code> variant will expand the <code><key></code>'s code once, so will be slightly quicker, but not change if you redefine <code><key></code>. The <code>edefault</code> on the other hand fully expands the <code><key></code>-code with <code><definition></code> as its argument inside of an <code>\edef</code>.</p>
<code>initial</code>	<code>initial <key> = {<value>}</code> <div style="text-align: right;"><code>protected long</code></div> <p>With <code>initial</code> you can set an initial <code><value></code> for an already defined argument taking <code><key></code>. It'll just call the key-macro of <code><key></code> and pass it <code><value></code>.</p>
<code>bool</code> <code>gbool</code> <code>boolTF</code> <code>gboolTF</code>	<code>bool <key> = <cs></code> <div style="text-align: right;"><code>protected long</code></div> <p>The <code><cs></code> should be a single control sequence, such as <code>\iff</code>. This will define <code><key></code> 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 <code><value></code> it'll have the same effect as if you use <code><key>=true</code>. <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 <code><cs></code> 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 <code><key>=<value></code> interface, but you could use <code>\newif</code> yourself. Even if the <code><key></code> 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 <key> = false</code> without an issue to change the default behaviour to execute the <code>false</code> choice).</p>

<u>store</u>	store $\langle key \rangle = \langle cs \rangle$	protected long
<u>estore</u>	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> .	
<u>gstore</u>		
<u>xstore</u>		
<u>int</u>	int $\langle key \rangle = \langle cs \rangle$	protected long
<u>eint</u>	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.	
<u>gint</u>		
<u>xint</u>		
<u>dimen</u>	dimen $\langle key \rangle = \langle cs \rangle$	protected long
<u>edimen</u>	The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . This is just like <code>int</code> but uses a <code>dimen</code> register, <code>\newdimen</code> and <code>\dimexpr</code> instead.	
<u>gdimen</u>		
<u>xdimen</u>		
<u>skip</u>	dimen $\langle key \rangle = \langle cs \rangle$	protected long
<u>eskip</u>	The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . This is just like <code>int</code> but uses a <code>skip</code> register, <code>\newskip</code> and <code>\glueexpr</code> instead.	
<u>gskip</u>		
<u>xskip</u>		
<u>toks</u>	dimen $\langle key \rangle = \langle cs \rangle$	protected long
<u>gtoks</u>	The $\langle cs \rangle$ should be a single control sequence, such as <code>\foo</code> . Store $\langle value \rangle$ inside of a <code>toks</code> -register. The <code>g</code> variants use <code>\global</code> , the <code>app</code> 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> .	
<u>apptoks</u>		
<u>gapptoks</u>		
<u>box</u>	box $\langle key \rangle = \langle cs \rangle$	protected long
<u>gbox</u>	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 <code>box</code> register. The boxes are colour safe. <code>expl\DEF</code> doesn't provide a <code>vbox</code> type.	
<u>meta</u>		
<u>meta</u>		
<u>meta</u>	meta $\langle key \rangle = \{\langle key \rangle = \langle value \rangle, \dots\}$	protected long
<u>meta</u>	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 <code>#1</code> . It works by calling a sub- <code>\ekvset</code> on the $\langle key \rangle = \langle value \rangle$ list, so a <code>set</code> key will only affect that $\langle key \rangle = \langle value \rangle$ list and not the current <code>\ekvset</code> .	
<u>nmeta</u>		
<u>nmeta</u>		
<u>nmeta</u>		
<u>nmeta</u>	nmeta $\langle key \rangle = \{\langle key \rangle = \langle value \rangle, \dots\}$	protected long
<u>nmeta</u>	This key type can set other keys, the difference to <code>meta</code> is, that this key doesn't take a value, so the $\langle key \rangle = \langle value \rangle$ list is static.	
<u>smeta</u>		
<u>smeta</u>		
<u>smeta</u>		
<u>smeta</u>	smeta $\langle key \rangle = \{\langle set \rangle\}\{\langle key \rangle = \langle value \rangle, \dots\}$	protected long
<u>smeta</u>	Yet another meta variant. An <code>smeta</code> key will take a $\langle value \rangle$ which you can access using <code>#1</code> , 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> .	
<u>smeta</u>		
<u>smeta</u>		
<u>smeta</u>		

snmeta	<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>`.

set	<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 `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>	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.

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
  ,int        keyE = \keyE
  ,dimen      keyF = \keyF
  ,skip       keyG = \keyG
  ,toks       keyH = \keyH
  ,default    keyH = \empty test
  ,box        keyI = \keyI
  ,qdefault   keyI = text
  ,choice     keyJ =
  {
    ,protected 1 = \texttt{a}
    ,2 = b
    ,3 = c
  }
}
```

```

,4 = d
,5 = e
}
,edefault keyJ = 2
,meta keyK = {keyA={#1},keyB=false}
,set setB = B
}

```

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{0.1}
16 \def\ekvdDate{2020-02-27}
```

(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` and `\ekvd@prot` to store whether a key should be defined an `\long` or `\protected`, and we have to clear them for every new key. By default they'll just be empty.

```
\ekvd@clear@prefixes
\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 $\langle key \rangle = \langle value \rangle$ 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.

```

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\def\ekvd@#1#2%
34 {%
35   \ekvd@clear@prefixes
36   \ekvd@ifspace{#1}%
37   {\ekvd@prefix\ekv@mark#1\ekv@stop{#2}}%
38   {\ekvd@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 TeX 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

```

129 \long\def\ekvd@t@initial#1#2%
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 }

```

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

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

```

\ekvd@t@boolTF
\ekvd@t@gboolTF
142 \protected\def\ekvd@type@bool#1#2#3#4#5#6#7%
143 {%
144     \ekvd@assert@filledarg{#1bool#2 #6}{#7}%
145     {%
146         \ekvd@newlet#7#5%
147         \ekvd@type@choice{#1bool#2}{#6}%
148         \protected\ekvdefNoVal\ekvd@set{#6}{#3\let#7#4}%
149         \protected\expandafter\def
150             \csname\ekvd@choice@name\ekvd@set{#6}{true}\endcsname
151             {#3\let#7#4}%
152         \protected\expandafter\def
153             \csname\ekvd@choice@name\ekvd@set{#6}{false}\endcsname
154             {#3\let#7#5}%
155     }%
156 }
157 \protected\def\ekvd@t@bool{\ekvd@type@bool{}{}{}\iftrue\iffalse}
158 \protected\def\ekvd@t@gbool{\ekvd@type@bool g{}\global\iftrue\iffalse}
159 \protected\def\ekvd@t@boolTF{\ekvd@type@bool{}{TF}}{\@firstoftwo\@secondoftwo}
160 \protected\def\ekvd@t@gboolTF
161     {\ekvd@type@bool g{TF}\global\@firstoftwo\@secondoftwo}

```

(End definition for \ekvd@type@bool 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`

```

162 \protected\def\ekvd@type@box#1#2#3#4%
163 {%
164   \ekvd@assert@filledarg{#1box #3}{#4}%
165   {%
166     \ekvd@newreg#4{box}%
167     \protected\ekvd@long\ekvdef\ekvd@set{#3}%
168       {#2\setbox#4\hbox{\begingroup##1\endgroup}}%
169   }%
170 }
171 \protected\def\ekvd@t@box{\ekvd@type@box{}}{}
172 \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
\ekvd@t@gtoks
173 \protected\def\ekvd@type@toks#1#2#3#4%
174 {%
175   \ekvd@assert@filledarg{#1toks #3}{#4}%
176   {%
177     \ekvd@newreg#4{toks}%
178     \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{##1}}%
179   }%
180 }
181 \protected\def\ekvd@t@toks{\ekvd@type@toks{}}{}
182 \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`
`\ekvd@t@gapptoks`

```

183 \protected\def\ekvd@type@apptoks#1#2#3#4%
184 {%
185   \ekvd@assert@filledarg{#1apptoks #3}{#4}%
186   {%
187     \ekvd@newreg#4{toks}%
188     \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4\expandafter{\the#4##1}}%
189   }%
190 }
191 \protected\def\ekvd@t@apptoks{\ekvd@type@apptoks{}}{}
192 \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 `<register>=<value>`.
`\ekvd@t@int`
`\ekvd@t@eint`
`\ekvd@t@gint`
`\ekvd@t@xint`
`\ekvd@t@dimen`
`\ekvd@t@edimen`
`\ekvd@t@gdimen`
`\ekvd@t@xdimen`
`\ekvd@t@skip`
`\ekvd@t@eskip`
`\ekvd@t@gskip`
`\ekvd@t@xskip`

```

193 \protected\def\ekvd@type@reg#1#2#3#4#5#6#7%
194 {%
195   \ekvd@assert@filledarg{#1 #6}{#7}%
196   {%
197     \ekvd@newreg#7{#2}%

```

```

198     \protected\ekvd@long\ekvdef\ekvd@set{#6}{#3#7=#4##1#5\relax}%
199   }%
200 }
201 \protected\def\ekvd@t@int{\ekvd@type@reg{int}{count}{}}{}
202 \protected\def\ekvd@t@eint{\ekvd@type@reg{eint}{count}{}}\numexpr\relax}
203 \protected\def\ekvd@t@gint{\ekvd@type@reg{gint}{count}}\global{}}{}
204 \protected\def\ekvd@t@xint{\ekvd@type@reg{xint}{count}}\global\numexpr\relax}
205 \protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{dimen}{}}{}{}
206 \protected\def\ekvd@t@edimen{\ekvd@type@reg{edimen}{dimen}{}}\dimexpr\relax}
207 \protected\def\ekvd@t@gdimen{\ekvd@type@reg{gdimen}{dimen}}\global{}}{}
208 \protected\def\ekvd@t@xdimen{\ekvd@type@reg{xdimen}{dimen}}\global\dimexpr\relax}
209 \protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{skip}{}}{}{}
210 \protected\def\ekvd@t@eskip{\ekvd@type@reg{eskip}{skip}{}}\glueexpr\relax}
211 \protected\def\ekvd@t@gskip{\ekvd@type@reg{gskip}{skip}}\global{}}{}
212 \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
 \ekvd@t@store also store # easily.

```

\ekvd@t@gstore 213 \protected\def\ekvd@type@store#1#2#3#4%
214   {%
215     \ekvd@assert@filledarg{#1store #3}{#4}%
216     {%
217       \unless\ifdefined#4\let#4\ekvd@empty\fi
218       \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{\unexpanded{##1}}}%
219     }%
220   }
221 \protected\def\ekvd@t@store{\ekvd@type@store{}\edef}
222 \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 223 \protected\def\ekvd@type@estore#1#2#3#4%
\ekvd@t@xstore 224   {%
225     \ekvd@assert@filledarg{#1store #3}{#4}%
226     {%
227       \ekvd@newlet#4\ekvd@empty
228       \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{##1}}%
229     }%
230   }
231 \protected\def\ekvd@t@estore{\ekvd@type@estore{e}\edef}
232 \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,
 \ekvd@type@meta@ with the same <set>.

```

\ekvd@t@meta 233 \protected\long\def\ekvd@type@meta#1#2#3#4#5%
\ekvd@t@nmeta 234   {%
235     \ekvd@assert@filledarg{#1meta #4}{#5}%
236     {%
237       \edef\ekvd@tmp{\ekvd@set}%
238       \expandafter\ekvd@type@meta@\expandafter{\ekvd@tmp}{#3}{#5}%

```

```

239         #2\ekvd@set{#4}\ekvd@tmp
240     }%
241 }
242 \protected\long\def\ekvd@type@meta@#1#2#3%
243 {%
244     \ekvd@prot\ekvd@long\def\ekvd@tmp#2{\ekvset{#1}{#3}}%
245 }
246 \protected\def\ekvd@t@meta{\ekvd@type@meta{}}\ekvlet{##1}}
247 \protected\def\ekvd@t@nmeta#1#2%
248 {%
249     \ekvd@assert@not@long{nmeta #1}%
250     \ekvd@type@meta n\ekvletNoVal{#1}{#2}%
251 }

```

(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
252 \protected\long\def\ekvd@type@smeta#1#2#3#4#5%
253 {%
254     \ekvd@assert@twoargs{s#1meta #4}{#5}%
255     {%
256         \expandafter\ekvd@type@smeta@\expandafter{\@secondoftwo#5}{#5}{#3}
257         #2\ekvd@set{#4}\ekvd@tmp
258     }%
259 }
260 \protected\long\def\ekvd@type@smeta@#1#2#3%
261 {%
262     \expandafter\ekvd@type@meta@\expandafter{\@firstoftwo#2}{#3}{#1}%
263 }
264 \protected\def\ekvd@t@smeta{\ekvd@type@smeta{}}\ekvlet{##1}}
265 \protected\def\ekvd@t@snmeta#1#2%
266 {%
267     \ekvd@assert@not@long{snmeta #1}%
268     \ekvd@type@smeta n\ekvletNoVal{#1}{#2}%
269 }

```

(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
270 \protected\def\ekvd@type@choice#1#2%
271 {%
272     \ekvd@assert@not@long{#1 #2}%
273     \ekvd@prot\edef\ekvd@tmp##1%
274     {%
275         \unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#2}{##1}}%
276     }%
277     \ekvlet\ekvd@set{#2}\ekvd@tmp
278 }

```

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

```

279 \protected\def\ekvd@populate@choice
280   {%
281     \ekvparse\ekvd@populate@choice@noarg\ekvd@populate@choice@
282   }
283 \protected\def\ekvd@populate@choice@noarg#1%
284   {%
285     \expandafter\ekvd@err@missing@definition\expandafter{\ekvd@set@choice : #1}%
286   }

```

\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.

```

287 \protected\long\def\ekvd@populate@choice@#1#2%
288   {%
289     \ekvd@clear@prefixes
290     \expandafter\ekvd@assert@arg\expandafter{\ekvd@set@choice : #1}{#2}%
291     {%
292       \ekvd@ifspace{#1}%
293       {\ekvd@choice@prefix\ekv@mark#1\ekv@stop}%
294       {%
295         \expandafter\def
296         \csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname
297       }%
298       {#2}%
299     }%
300   }
301 \protected\def\ekvd@choice@prefix#1
302   {%
303     \ekv@strip{#1}\ekvd@choice@prefix@\ekv@mark
304   }
305 \protected\def\ekvd@choice@prefix@#1#2\ekv@stop
306   {%
307     \ekv@ifdefined{\ekvd@choice@p@#1}%
308     {%
309       \csname \ekvd@choice@p@#1\endcsname
310       \ekvd@ifspace{#2}%
311       {\ekvd@choice@prefix#2\ekv@stop}%
312       {%
313         \ekvd@prot\expandafter\def
314         \csname
315           \ekv@strip{#2}{\ekvd@choice@name\ekvd@set\ekvd@set@choice}%
316         \endcsname
317       }%
318     }%
319     {\ekvd@err@undefined@prefix{#1}\@gobble}%
320   }
321 \protected\def\ekvd@choice@p@protected{\let\ekvd@prot\protected}
322 \let\ekvd@choice@p@protect\ekvd@choice@p@protected
323 \protected\def\ekvd@choice@p@long\ekvd@ifspace#1%
324   {%
325     \expandafter\ekvd@choice@p@long@\expandafter{\ekv@gobble@mark#1}%

```

```

326     \ekvd@ifspace{#1}%
327 }
328 \protected\def\ekvd@choice@p@long@#1%
329 {%
330     \expandafter\ekvd@err@no@long\expandafter
331     {\ekvd@set@choice : long #1}%
332 }

```

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.

```

333 \protected\long\def\ekvd@t@choice#1#2%
334 {%
335     \ekvd@assert@arg{choice #1}{#2}%
336     {%
337         \ekvd@type@choice{choice}{#1}%
338         \def\ekvd@set@choice{#1}%
339         \ekvd@populate@choice{#2}%
340     }%
341 }

```

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

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.

```

342 \def\ekvd@h@choice#1%
343 {%
344     \expandafter\ekvd@h@choice@
345     \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname
346     {#1}%
347 }
348 \def\ekvd@h@choice@#1#2%
349 {%
350     \ifx#1\relax
351         \ekvd@err@choice@invalid{#2}%
352         \expandafter\@gobble
353     \fi
354     #1%
355 }

```

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

2.2.3 Tests

`\ekvd@noarg@mark` This macro serves as a flag for the case that no <value> was specified for a key. As such it is not a test, but exists only for some tests.

```

356 \def\ekvd@noarg@mark{\ekvd@noarg@mark}

```

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

`\ekvd@fi@firstoftwo` While we can reuse many of the internals of `expkv` the specific case for this branch wasn't needed by `expkv` and hence isn't defined. We'll need it, so we define it.

```

357 \long\def\ekvd@fi@firstoftwo\fi\@secondoftwo#1#2{\fi#1}

(End definition for \ekvd@fi@firstoftwo.)

```

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

```

358 \protected\def\ekvd@newlet#1#2%
359   {%
360     \unless\ifdefined#1\let#1#2\fi
361   }
362 \protected\def\ekvd@newreg#1#2%
363   {%
364     \unless\ifdefined#1\csname new#2\endcsname#1\fi
365   }

(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,
`\ekvd@ifnottwoargs` in the case that there are fewer tokens than two in the argument, only macros will be
`\ekvd@ifempty@gtwo` gobbled that are needed for the true branch, which doesn't hurt, and if there are more
this will not be empty.

```

366 \long\def\ekvd@assert@twoargs#1#2%
367   {%
368     \ekvd@ifnottwoargs{#2}%
369     {\ekvd@err@missing@definition{#1}}%
370   }
371 \long\def\ekvd@ifnottwoargs#1%
372   {%
373     \ekvd@ifempty@gtwo#1\ekv@ifempty@B
374     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
375   }
376 \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@arg` The test for an argument is just an `\ifx` comparison with our `noarg@mark`.
`\ekvd@ifnoarg`

```

377 \long\def\ekvd@assert@arg#1#2%
378   {%
379     \ekvd@ifnoarg{#2}%
380     {\ekvd@err@missing@definition{#1}}%
381   }
382 \long\def\ekvd@ifnoarg#1%
383   {%
384     \ifx\ekvd@noarg@mark#1%
385     \ekvd@fi@firstoftwo
386     \fi
387     \@secondoftwo
388   }

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

```

```

\ekvd@assert@filledarg
\ekvd@ifnoarg@or@empty
389 \long\def\ekvd@assert@filledarg#1#2%
390 {%
391     \ekvd@ifnoarg@or@empty{#2}%
392     {\ekvd@err@missing@definition{#1}}%
393 }
394 \long\def\ekvd@ifnoarg@or@empty#1%
395 {%
396     \ekvd@ifnoarg{#1}%
397     \@firstoftwo
398     {\ekv@ifempty{#1}}%
399 }

```

(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.

```

400 \long\def\ekvd@assert@not@long#1%
401 {%
402     \ifx\ekvd@long\long\ekvd@err@no@long{#1}\fi
403 }
404 \long\def\ekvd@assert@not@protected#1%
405 {%
406     \ifx\ekvd@prot\protected\ekvd@err@no@protected{#1}\fi
407 }

```

(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.

```

408 \long\def\ekvd@ifspace#1%
409 {%
410     \ekvd@ifspace@#1 \ekv@ifempty@B
411     \ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
412 }
413 \long\def\ekvd@ifspace@#1 % keep this space
414 {%
415     \ekv@ifempty@\ekv@ifempty@A
416 }

```

(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:

```

417 \protected\def\ekvd@err@missing@definition#1%
418     {\errmessage{expkv-def Error: Missing definition for key '\unexpanded{#1}'} }
419 \protected\def\ekvd@err@missing@prefix#1%
420     {\errmessage{expkv-def Error: Missing prefix for key '\unexpanded{#1}'} }
421 \protected\def\ekvd@err@undefined@prefix#1%

```

```

422 {\errmessage{expkv-def Error: Undefined prefix '\unexpanded{#1}'}}
423 \protected\def\ekvd@err@undefined@key#1%
424 {\errmessage{expkv-def Error: Undefined key '\unexpanded{#1}'}}
425 \protected\def\ekvd@err@no@protected#1%
426 {%
427   \errmessage
428     {expkv-def Error: prefix 'protected' not accepted for '\unexpanded{#1}'}%
429 }
430 \protected\def\ekvd@err@no@long#1%
431 {%
432   \errmessage
433     {expkv-def Error: prefix 'long' not accepted for '\unexpanded{#1}'}%
434 }

```

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

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

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.

```

435 \def\ekvd@err@choice@invalid#1%
436 {%
437   \ekvd@err@choice@invalid@#1\ekv@stop
438 }
439 \begingroup
440 \catcode40=8
441 \catcode41=8
442 \@firstofone{\endgroup
443 \def\ekvd@choice@name#1#2#3%
444 {%
445   ekvd#1(#2)#3%
446 }
447 \def\ekvd@err@choice@invalid@ ekvd#1(#2)#3\ekv@stop%
448 {%
449   \ekvd@err{invalid choice '#3' ('#2', set '#1')}%
450 }
451 }
452 \begingroup
453 \edef\ekvd@err
454 {%
455   \endgroup
456   \unexpanded{\long\def\ekvd@err}##1%
457   {%
458     \unexpanded{\expandafter\ekv@err@\@firstofone}%
459     {\expandafter\noexpand\csname ! expkv-def Error:\endcsname ##1.}%
460     \unexpanded{\ekv@stop}%
461   }%
462 }
463 \ekvd@err

```

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

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

464 \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	
apptoks	4
B	
bool	3
boolTF	3
box	4
C	
choice	5
code	3
D	
default	3
dimen	4
E	
ecode	3
edefault	3
edimen	4
eint	4
\ekvchangeset	68, 72
\ekvdDate	2, 5, 9, <u>15</u>
\ekvdef	167, 178, 188, 198, 218, 228
\ekvdefinekeys	2, <u>27</u>
\ekvdefNoVal	68, 72, 148
\ekvdVersion	2, 5, 9, <u>15</u>
\ekvifdefined	100, 119, 133
\ekvlet	91, 246, 264, 277
\ekvletNoVal	81, 108, 124, 250, 268
\ekvparse	30, 281
\ekvset	244
enoval	3
eskip	4
estore	4
G	
gapptoks	4
gbool	3
gboolTF	3
gbox	4
gdimen	4
gint	4
gskip	4
gstore	4
gtoks	4
I	
initial	3
L	
long	3
M	
meta	4
N	
nmeta	4
noval	3
P	
protect	3
protected	3
Q	
qdefault	3
S	
set	5
skip	4
smeta	4
snmeta	5
store	4
T	
TeX and L ^A T _E X commands:	
\@firstofone	442, 458
\@firstoftwo	159, 161, 262, 374, 397, 411
\@gobble	48, 57, 319, 352
\@secondoftwo	159, 161, 256, 357, 387
\ekv@err@	458
\ekv@gobble@mark	56, 325
\ekv@ifdefined	43, 46, 307
\ekv@ifempty	70, 398
\ekv@ifempty@	376, 415
\ekv@ifempty@A	374, 376, 411, 415
\ekv@ifempty@B	373, 374, 410, 411
\ekv@ifempty@false	374, 411
\ekv@mark	37, 40, 293, 303
\ekv@name	106, 123, 137
\ekv@stop	37,
	41, 54, 293, 305, 311, 437, 447, 460
\ekv@strip	40, 44, 303, 315
\ekvd@	30, <u>32</u>
\ekvd@assert@arg	...
	77, 88, 98, 117, 131, 290, 335, <u>377</u>
\ekvd@assert@filledarg	144,
	164, 175, 185, 195, 215, 225, 235, <u>389</u>

\ekvd@assert@not@long	65,	\ekvd@populate@choice@noarg	270
79, 102, 121, 135, 249, 267, 272, 400		\ekvd@prefix	37, 40, 54
\ekvd@assert@not@protected		\ekvd@prefix@	40
.....	66, 136, 400	\ekvd@prefix@after@p	51, 60, 61
\ekvd@assert@twoargs	254, 366	\ekvd@prot	20, 61,
\ekvd@choice@name		80, 90, 103, 122, 244, 273, 313, 321, 406	
.....	150, 153, 275, 296, 315, 435	\ekvd@set	29, 68, 72, 81, 91, 100, 106,
\ekvd@choice@p@long	270	108, 119, 123, 124, 133, 137, 148,	
\ekvd@choice@p@long@	270	150, 153, 167, 178, 188, 198, 218,	
\ekvd@choice@p@protect	270	228, 237, 239, 257, 275, 277, 296, 315	
\ekvd@choice@p@protected	270	\ekvd@set@choice	
\ekvd@choice@prefix	270	285, 290, 296, 315, 331, 338
\ekvd@choice@prefix@	270	\ekvd@t@apptoks	183
\ekvd@clear@prefixes	20, 35, 289	\ekvd@t@bool	142
\ekvd@empty	20, 217, 227	\ekvd@t@boolTF	142
\ekvd@err	435	\ekvd@t@box	162
\ekvd@err@choice@invalid ..	351, 435	\ekvd@t@choice	270
\ekvd@err@choice@invalid@	435	\ekvd@t@code	86
\ekvd@err@missing@definition ..		\ekvd@t@default	96
.....	71, 285, 369, 380, 392, 417	\ekvd@t@dimen	193
\ekvd@err@missing@prefix ...	56, 417	\ekvd@t@ecode	86
\ekvd@err@no@long	330, 402, 417	\ekvd@t@edefault	115
\ekvd@err@no@protected	406, 417	\ekvd@t@edimen	193
\ekvd@err@undefined@key		\ekvd@t@eint	193
.....	110, 126, 139, 417	\ekvd@t@enoval	75
\ekvd@err@undefined@prefix		\ekvd@t@eskip	193
.....	48, 319, 417	\ekvd@t@estore	223
\ekvd@fi@firstoftwo	357, 385	\ekvd@t@gapptoks	183
\ekvd@h@choice	275, 342	\ekvd@t@gbool	142
\ekvd@h@choice@	342	\ekvd@t@gboolTF	142
\ekvd@ifempty@gtwo	366	\ekvd@t@gbox	162
\ekvd@ifnoarg	67, 377, 396	\ekvd@t@gdimen	193
\ekvd@ifnoarg@or@empty	389	\ekvd@t@gint	193
\ekvd@ifnottwoargs	366	\ekvd@t@gskip	193
\ekvd@ifspace		\ekvd@t@gstore	213
.....	36, 53, 292, 310, 323, 326, 408	\ekvd@t@gtoks	173
\ekvd@ifspace@	408	\ekvd@t@initial	129
\ekvd@l@long	20, 60, 90,	\ekvd@t@int	193
167, 178, 188, 198, 218, 228, 244, 402		\ekvd@t@meta	233
\ekvd@missing@prefix	38	\ekvd@t@nmeta	233
\ekvd@newlet	146, 227, 358	\ekvd@t@noval	75
\ekvd@newreg ...	166, 177, 187, 197, 358	\ekvd@t@qdefault	96
\ekvd@noarg	30, 32	\ekvd@t@set	63
\ekvd@noarg@mark	32, 356, 384	\ekvd@t@skip	193
\ekvd@p@long	60	\ekvd@t@smeta	252
\ekvd@p@protect	60	\ekvd@t@snmeta	252
\ekvd@p@protected	60	\ekvd@t@store	213
\ekvd@populate@choice	270		
\ekvd@populate@choice@	270		

\ekvd@t@toks	<u>173</u>	\ekvd@type@meta	<u>233</u>
\ekvd@t@xdimen	<u>193</u>	\ekvd@type@meta@	<u>233</u> , <u>262</u>
\ekvd@t@xint	<u>193</u>	\ekvd@type@noval	<u>75</u>
\ekvd@t@xskip	<u>193</u>	\ekvd@type@reg	<u>193</u>
\ekvd@t@xstore	<u>223</u>	\ekvd@type@smeta	<u>252</u>
\ekvd@tmp	2,	\ekvd@type@smeta@	<u>252</u>
80, 81, 90, 91, 103, 108, 122, 124,		\ekvd@type@store	<u>213</u>
237, 238, 239, 244, 257, 273, 277, 464		\ekvd@type@toks	<u>173</u>
\ekvd@type@apptoks	<u>183</u>	toks	4
\ekvd@type@bool	<u>142</u>		
\ekvd@type@box	<u>162</u>		
\ekvd@type@choice	147, <u>270</u>		
\ekvd@type@code	<u>86</u>		
\ekvd@type@default	<u>96</u>		
\ekvd@type@estore	<u>223</u>		

X

xdimen	4
xint	4
xskip	4
xstore	4