

# The xkeyval package \*

Hendri Adriaens  
<http://stuwwww.uvt.nl/~hendri>

v2.6a (2008/08/13)

## Abstract

This package is an extension of the keyval package and offers more flexible macros for defining and setting keys. The package provides a pointer and a preset system. Furthermore, it supplies macros to allow class and package options to contain options of the `key=value` form. A  $\LaTeX$  kernel patch is provided to avoid premature expansions of macros in class or package options. A specialized system for setting PSTricks keys is provided by the `pst-xkey` package.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>	<b>10</b>	<b>Category codes</b>	<b>22</b>
<b>2</b>	<b>Loading xkeyval</b>	<b>3</b>	<b>11</b>	<b>Known issues</b>	<b>23</b>
<b>3</b>	<b>Defining and managing keys</b>	<b>3</b>	<b>12</b>	<b>Additional packages</b>	<b>24</b>
3.1	Ordinary keys . . . . .	4	12.1	xkvview . . . . .	24
3.2	Command keys . . . . .	4	12.2	xkvltxp . . . . .	25
3.3	Choice keys . . . . .	5	12.3	pst-xkey . . . . .	26
3.4	Boolean keys . . . . .	7	<b>13</b>	<b>Examples and documentation</b>	<b>27</b>
3.5	Checking keys . . . . .	8	<b>14</b>	<b>Implementation</b>	<b>28</b>
3.6	Disabling keys . . . . .	8	14.1	xkeyval.tex . . . . .	28
<b>4</b>	<b>Setting keys</b>	<b>8</b>	14.2	xkeyval.sty . . . . .	55
4.1	The user interface . . . . .	8	14.3	keyval.tex . . . . .	58
4.2	A few details . . . . .	10	14.4	xkvtxhdr.tex . . . . .	59
<b>5</b>	<b>Pointers</b>	<b>11</b>	14.5	xkvview.sty . . . . .	61
5.1	Saving values . . . . .	11	14.6	xkvltxp.sty . . . . .	64
5.2	Using saved values . . . . .	13	14.7	pst-xkey.tex . . . . .	66
<b>6</b>	<b>Presetting keys</b>	<b>15</b>	14.8	pst-xkey.sty . . . . .	67
<b>7</b>	<b>Package option processing</b>	<b>17</b>	<b>References</b>	<b>67</b>	
<b>8</b>	<b>List of macro structures</b>	<b>19</b>	<b>Acknowledgements</b>	<b>68</b>	
<b>9</b>	<b>Warnings and errors</b>	<b>21</b>	<b>Version history</b>	<b>68</b>	
			<b>Index</b>	<b>69</b>	

---

\*This package can be downloaded from the CTAN mirrors: `/macros/latex/contrib/xkeyval`. See `xkeyval.dtx` for information on installing xkeyval into your  $\TeX$  or  $\LaTeX$  distribution and for the license of this package.

# 1 Introduction

This package is an extension of the `keyval` package by David Carlisle [3] and offers more flexible and robust macros for defining and setting keys. Using keys in macro definition has the advantage that the 9 arguments maximum can easily be avoided and that it reduces confusion in the syntax of your macro when compared to using a lot of (optional) arguments. Compare for instance the following possible syntaxes of the macro `\mybox` which might for instance use its arguments to draw some box containing text.

```
\mybox[5pt][20pt]{some text}[red][white][blue]
\mybox[text=red,background=white,frame=blue,left=5pt,right=20pt]{some text}
```

Notice that, to be able to specify the frame color in the first example, the other colors need to be specified as well. This is not necessary in the second example and these colors can get preset values. The same thing holds for the margins.

The idea is that one first defines a set of keys using the tools presented in section 3 in the document preamble or in a package or class. These keys can perform a function with the user input. The way to submit user input to these key macros, is by using one of the user interfaces described in sections 4, 5 and 6. The main user interface is provided by the `\setkeys` command. Using these interfaces, one can simplify macro syntax and for instance define the `\mybox` macro above as follows.

```
\define@key{mybox}{left}{\setlength\myleft{#1}}
\define@key{mybox}{background}{\def\background{#1}}
% and some other keys
\def\mybox{\@ifnextchar[\@mybox{\@mybox[]}}
\def\@mybox[#1]#2{%
  \setkeys{mybox}{#1}%
  % some operations to typeset #2
}
```

Notice that the combination of the two definitions `\mybox` and `\@mybox` can be replaced by `\newcommand\mybox[2] [] { . . . }` when using  $\LaTeX$ .

Both keys defined using the `keyval` and `xkeyval` can be set by this package. The `xkeyval` macros allow for scanning multiple sets (called ‘families’) of keys. This can, for instance, be used to create local families for macros and environments which may not access keys meant for other macros and environments, while at the same time, allowing the use of a single command to (pre)set all of the keys from the different families globally.

The package is compatible to plain  $\TeX$  and redefines several `keyval` macros to provide an easy way to switch between using `keyval` and `xkeyval`. This might be useful for package writers that cannot yet rely on the availability of `xkeyval` in a certain distribution. After loading `xkeyval`, loading `keyval` is prevented to make sure that the extended macros of `xkeyval` will not be redefined. Some internal `keyval` macros are supplied in `keyval.tex` to guarantee compatibility to packages that use those macros. Section 11 provides more information about this issue.

The organization of this documentation is as follows. Section 2 discusses how to load `xkeyval` and what the package does when it is loaded. Section 3 will discuss the macros available to define and manage keys. Section 4 will continue with describing the macros that can set the keys. Section 5 explains special syntax which will allow saving and copying key values. In section 6, the preset system will be introduced. Section 10 will explain how `xkeyval` protects itself for catcode changes of the comma and the equality sign by other packages. The `xkeyval` package also provides commands

to declare and process class and package options that can take values. These will be discussed in section 7. Section 8 provides an overview of structures used to create `xkeyval` internal macros used for keys, values, presets, etcetera. Sections 9 and 11 discuss feedback that `xkeyval` might give and known issues, respectively. Section 12 discusses several additional packages that come with the `xkeyval` bundle. Section 12.1 presents a viewer utility which produces overviews of defined keys. An extension of the  $\text{\LaTeX}2_{\epsilon}$  kernel with respect to the class and package options system is discussed in section 12.2. This extension provides a way to use expandable macros in package options. Section 12.3 presents the `pst-xkey` package, which provides an options system based on `xkeyval`, but which is specialized in setting `PSTricks` keys.

Throughout this documentation, you will find some examples with a short description. More examples can be found in the example files that come with this package. See section 13 for more information. This section also provides the information how to generate the full documentation from the source.

## 2 Loading `xkeyval`

To load the `xkeyval` package,<sup>1</sup> plain  $\text{\TeX}$  users do `\input xkeyval`.  $\text{\LaTeX}$  users do one of the following: `\usepackage{xkeyval}` or `\RequirePackage{xkeyval}`. The package does not have options. It is mandatory for  $\text{\LaTeX}$  users to load `xkeyval` at any point after the `\documentclass` command. Loading `xkeyval` from the class which is the document class itself is possible. The package will use the  $\epsilon$ - $\text{\TeX}$  engine when available. In particular, `\ifcstype` is used whenever possible to avoid filling  $\text{\TeX}$ 's hash tables with useless entries, for instance when searching for keys in families.

If `xkeyval` is loaded by `\RequirePackage` or `\usepackage`, the package performs two action immediately. These require `xkeyval` to be loaded at any point after `\documentclass` or by the document class itself.

First, it retrieves the document class of the document at hand and stores that (including the class extension) into the following macro.

```
\XKV@documentclass
```

`\XKV@documentclass` This macro could, for instance, contain `article.cls` and can be useful when using `\ProcessOptionsX*` in a class. See page 18.

Secondly, the global options submitted to the `\documentclass` command and stored by  $\text{\LaTeX}$  in `\@classoptionslist` are copied to the following macro.

```
\XKV@classoptionslist
```

`\XKV@classoptionslist` This macro will be used by `\ProcessOptionsX`. Options containing an equality sign are deleted from the original list in `\@classoptionslist` to avoid packages, which do not use `xkeyval` and which are loaded later, running into problems when trying to copy global options using  $\text{\LaTeX}$ 's `\ProcessOptions`.

## 3 Defining and managing keys

This section discusses macros to define keys and some tools to manage keys. A useful extension to `xkeyval` is the `xkvview` package. This packages defines commands to generate overviews of defined keys. See section 12.1 for more information.

<sup>1</sup>The `xkeyval` package consists of the files `xkeyval.tex`, `xkeyval.sty`, `keyval.tex`, `xkvtxhdr.tex`.

### 3.1 Ordinary keys

This section describes how to define ordinary keys.

```
\define@key [<prefix>] {<family>} {<key>} [<default>] {<function>}
```

\define@key

This defines a macro of the form  $\langle prefix \rangle @ \langle family \rangle @ \langle key \rangle$  with one argument holding  $\langle function \rangle$ . The default value for  $\langle prefix \rangle$  is KV. This is the standard throughout the package to simplify mixing keyval and xkeyval keys. When  $\langle key \rangle$  is used in a `\setkeys` command (see section 4) containing `key=value`, the macro  $\langle prefix \rangle @ \langle family \rangle @ \langle key \rangle$  receives value as its argument and will be executed. The argument can be accessed by  $\langle function \rangle$  by using #1 inside the function.

```
\define@key{family}{key}{The input is: #1}
```

xkeyval will generate an error when the user omits `=value` for a key in the options list of `\setkeys` (see section 4). To avoid this, the optional argument can be used to specify a default value.

```
\define@key{family}{key}[none]{The input is: #1}
```

This will additionally define a macro  $\langle prefix \rangle @ \langle family \rangle @ \langle key \rangle @ default$  as a macro with no arguments and definition  $\langle prefix \rangle @ \langle family \rangle @ \langle key \rangle \{ none \}$  which will be used when `=value` is missing for key in the options list. So, the last example comes down to doing

```
\def\KV@family@key#1{The input is: #1}
\def\KV@family@key@default{\KV@family@key{none}}
```

When  $\langle prefix \rangle$  is specified and empty, the macros created by `\define@key` will have the form  $\langle family \rangle @ \langle key \rangle$ . When  $\langle family \rangle$  is empty, the resulting form will be  $\langle prefix \rangle @ \langle key \rangle$ . When both  $\langle prefix \rangle$  and  $\langle family \rangle$  are empty, the form is  $\langle key \rangle$ . This combination of prefix and family will be called the header. The rules to create the header will be applied to all commands taking (optional) prefix and family arguments.

The intended use for  $\langle family \rangle$  is to create distinct sets of keys. This can be used to avoid a macro setting keys meant for another macro only. The optional  $\langle prefix \rangle$  can be used to identify keys specifically for your package. Using a package specific prefix reduces the probability of multiple packages defining the same key macros. This optional argument can also be used to set keys of some existing packages which use a system based on keyval.<sup>2</sup>

We now define some keys to be used in examples throughout this documentation.

```
\define@key [my] {familya} {keya} [default] {#1}
\define@key [my] {familya} {keyb} {#1}
\define@key [my] {familyb} {keyb} {#1}
\define@key [my] {familya} {keyc} {#1}
```

### 3.2 Command keys

Command keys are specialized keys that, before executing any code, save the user input to a macro.

<sup>2</sup>Like PSTricks, which uses a system originating from keyval, but which has been modified to use no families and pset as prefix.

```
\define@cmdkey [<prefix>]{<family>}[<mp>]{<key>}[<default>]{<function>}
```

`\define@cmdkey` This has the effect of defining a key macro of the form `\<prefix>@<family>@<key>` that, when used, first saves the user input to a macro of the form `\<mp><key>` and then executes `<function>`. `<mp>` is the macro prefix. If `<mp>` is not specified, the usual combination of `<prefix>` and `<family>`, together with the extra prefix `cmd`, will be used to create the macro prefix, namely `\cmd<prefix>@<family>@`.<sup>3</sup> The two keys in the following example hence do exactly the same thing.<sup>4</sup>

```
\define@cmdkey{fam}{key}[none]{value: \cmdKV@fam@key}
\define@key{fam}{key}[none]{\def\cmdKV@fam@key{#1}value: \cmdKV@fam@key}
```

The value `none` is again the default value that will be submitted to the key macro when the user didn't supply a value. (See also section 3.1 for more information.)

The following two lines also implement a key with the same key macro.

```
\define@cmdkey{fam}[my@]{key}[none]{value: \my@key}
\define@key{fam}{key}[none]{\def\my@key{#1}value: \my@key}
```

Note that the key macro itself in the examples above is still `\KV@fam@key`, just as in the previous example.

A lot of packages define keys that only save their value to a macro so that it can be used later. Using the macro above, one can save some tokens in the package. Some more tokens can be saved by using the following macro.

```
\define@cmdkeys [<prefix>]{<family>}[<mp>]{<keys>}[<default>]
```

`\define@cmdkeys` This repeatedly calls (an internal of) `\define@cmdkey` for all keys in the list of `<keys>`. Note that it is not possible to specify a custom key function for the keys created by this command. The only function of those keys is to save user input in a macro. The first line and the last two lines of the following example create keys with the same internal key macro.

```
\define@cmdkeys{fam}[my@]{keya,keyb}[none]
\define@key{fam}{keya}[none]{\def\my@keya{#1}}
\define@key{fam}{keyb}[none]{\def\my@keyb{#1}}
```

### 3.3 Choice keys

Choice keys allow only a limited number of different values for user input. These keys are defined as follows.

```
\define@choicekey [<pre>]{<fam>}{<key>}[<bin>]{<al>}[<dft>]{<func>}
\define@choicekey* [<pre>]{<fam>}{<key>}[<bin>]{<al>}[<dft>]{<func>}
```

`\define@choicekey`  
`\define@choicekey*` The keys work the same as ordinary keys, except that, before executing anything, it is verified whether the user input `#1` is present in the comma separated list `<al>`. The starred version first converts the input in `#1` and `<al>` to lowercase before performing the check. If the input is not allowed, an error is produced and the key macro `<func>`

<sup>3</sup>Remember that some rules are applied when creating the header, the combination of `<prefix>` and `<header>`. See section 3.1.

<sup>4</sup>Notice however, that the first key will be listed as a 'command key' by `xkview` and the second as an 'ordinary key'. See section 12.1.

will not be executed. If the input is allowed, the key macro  $\langle func \rangle$  will be executed.  $\langle dft \rangle$  is submitted to the key macro when the user didn't supply a value for the key. (See also section 3.1.)

The optional  $\langle bin \rangle$  should contain either one or two control sequences (macros). The first one will be used to store the user input used in the input check (hence, in lowercase when the starred version was used). The original user input will always be available in #1. The second (if present) will contain the number of the input in the  $\langle al \rangle$  list, starting from 0. The number will be set to -1 if the input was not allowed. The number can, for instance, be used in a  $\backslash ifcase$  statement in  $\langle func \rangle$ .

```
\define@choicekey*{fam}{align}[\val\nr]{left,center,right}{%
\ifcase\nr\relax
\raggedright
\or
\centering
\or
\raggedleft
\fi
}
```

The example above only allows input values `left`, `center` and `right`. Notice that we don't need a `\else` case in the key macro above as the macro will not be executed when the input was not allowed.

```
\define@choicekey+[\pre]{\fam}{\key}{\bin}{\al}{\dft}{\f1}{\f2}
\define@choicekey*+[\pre]{\fam}{\key}{\bin}{\al}{\dft}{\f1}{\f2}
```

$\backslash define@choicekey+$   
 $\backslash define@choicekey*+$

These macros operate as their counterparts without the `+`, but allow for specifying two key macros.  $\langle f1 \rangle$  will be executed when the input was correct and  $\langle f2 \rangle$  will be executed when the input was not allowed. Again, the starred version executes the check after converting user input and  $\langle al \rangle$  to lowercase.

```
\define@choicekey*+{fam}{align}[\val\nr]{left,center,right}{%
\ifcase\nr\relax
\raggedright
\or
\centering
\or
\raggedleft
\fi
}{%
\PackageWarning{mypack}{erroneous input ignored}%
}
```

The example above defines a key that is similar as the one in the previous example, but when input is not allowed, it will not generate a standard `xkeyval` warning, but will execute a custom function, which, in this case, generates a warning.

```
\XKV@cc[\bin]{\input}{\al}{\func}
\XKV@cc*[\bin]{\input}{\al}{\func}
\XKV@cc+[\bin]{\input}{\al}{\func1}{\func2}
\XKV@cc*+[\bin]{\input}{\al}{\func1}{\func2}
```

$\backslash XKV@cc$   
 $\backslash XKV@cc*$   
 $\backslash XKV@cc+$   
 $\backslash XKV@cc*+$

Choice keys work by adding (an internal version<sup>5</sup> of) the  $\backslash XKV@cc$  macro to key macros. This macro has similar arguments as the  $\backslash define@choicekey$  macro and

<sup>5</sup>See section 14 for details of the implementation of choice keys.

the optional `*` and `+` have the same meaning. `<input>` holds the input that should be checked, namely, whether it is (in lowercase if `*` was used) in the list `<al>`. One can use this macro to create custom choice keys. See an example below.

```
\define@key{fam}{key}{%
  I will first check your input, please wait.\
  \XKV@cc*[\val]{#1}{true,false}{%
    The input \val\ was correct, we proceed.\
  }{%
    The input \val\ was incorrect and was ignored.\
  }%
}
I finished the input check.
}
```

Try to find out why this key cannot be defined with `\define@boolkey` which is introduced in the next section.

### 3.4 Boolean keys

This section describes boolean keys which can be either true or false. A boolean key is a special version of a choice key (see section 3.3), where `<al>` takes the value `true`, `false` and comparisons are always done in lowercase mode (so, `True` is allowed input).

```
\define@boolkey[<pre>]{<fam>}[<mp>]{<key>}[<default>]{<func>}
\define@boolkey+ [<pre>]{<fam>}[<mp>]{<key>}[<default>]{<func1>}{<func2>}
```

```
\define@boolkey
\define@boolkey+
```

This creates a boolean of the form `\if<pre>@<family>@<key>`<sup>3,6</sup> if `<mp>` is not specified, using `\newif`<sup>7</sup> (which initiates the conditional to `\iffalse`) and a key macro of the form `\<pre>@<family>@<key>` which first checks the validity of the user input. If the input was valid, it uses it to set the boolean and afterwards, it executes `<func>`. If the input was invalid, it will not set the boolean and `xkeyval` will generate an error. If `<mp>` is specified, it will create boolean of the form `\if<mp><key>` (compare to command keys in section 3.2). The value `<default>` will be used by the key macro when the user didn't submit a value to the key. (See also section 3.1.)

If the `+` version of the macro is used, one can specify two key macros. If user input is valid, the macro will set the boolean and executes `<func1>`. Otherwise, it will not set the boolean and execute `<func2>`.

```
\define@boolkey{fam}[my@]{frame}{}
\define@boolkey+{fam}{shadow}{%
  \ifKV@fam@shadow
    \PackageInfo{mypack}{turning shadows on}%
  \else
    \PackageInfo{mypack}{turning shadows off}%
  \fi
}%
\PackageWarning{mypack}{erroneous input ignored}%
}
```

<sup>6</sup>When you want to use this macro directly, either make sure that neither of the input parameters contain characters with a catcode different from 11 (hence no `-` for instance), reset the catcode of the offending characters internally to 11 or use `\csname... \endcsname` to construct macro names, (for instance, `\csname ifpre@some-fam@key\endcsname`). See for more information section 8.

<sup>7</sup>The  $\LaTeX$  implementation `\newif` is used because it can be used in the replacement text of a macro, whereas the plain  $\TeX$  `\newif` is defined `\outer`.

The first example creates the boolean `\ifmy@frame` and defines the key macro `\KV@fam@frame` to only set the boolean (if input is correct). The second key informs the user about changed settings or produces a warning when input was incorrect.

One can also define multiple boolean keys with a single command.

```
\define@boolkeys [<pre>]{<fam>}[<mp>]{<keys>}[<default>]
```

`\define@boolkeys`

This macro creates a boolean key for every entry in the list `<keys>`. As with the command `\define@cmdkeys`, the individual keys cannot have a custom function. The boolean keys created with this command are only meant to set the state of the boolean using the user input. Concluding,

```
\define@boolkeys{fam}[my@]{keya,keyb,keyc}
```

is an abbreviation for

```
\define@boolkey{fam}[my@]{keya}{-}
\define@boolkey{fam}[my@]{keyb}{-}
\define@boolkey{fam}[my@]{keyc}{-}
```

### 3.5 Checking keys

```
\key@ifundefined [<prefix>]{<families>}{<key>}{<undefined>}{<defined>}
```

`\key@ifundefined`

This macro executes `<undefined>` when `<key>` is not defined in a family listed in `<families>` using `<prefix>` (which is KV by default) and `<defined>` when it is. If `<defined>` is executed, `\XKV@tfam` holds the first family in the list `<families>` that holds `<key>`. If `<undefined>` is executed, `\XKV@tfam` contains the last family of the list `<families>`.

```
\key@ifundefined [my]{familya,familyb}{keya}{'keya' not defined}{'keya' defined}
```

This example results in `'keya'` defined and `\XKV@tfam` holds `familya`.

### 3.6 Disabling keys

It is also possible to disable keys after use as to prevent the key from being used again.

```
\disable@keys [<prefix>]{<family>}{<keys>}
```

`\disable@keys`

When you disable a key, the use of this key will produce a warning in the log file. Disabling a key that hasn't been defined will result in an error message.

```
\disable@keys [my]{familya}{keya,keyb}
```

This would make `keya` and `keyb` produce a warning when one tries to set these keys.

## 4 Setting keys

### 4.1 The user interface

This section describes the available macros for setting keys. All of the macros in this section have an optional argument `<prefix>` which determines part of the form of the keys that the macros will be looking for. See section 3. This optional argument takes the value KV by default.



```
\setkeys [prefix] {families} [na] {keys}
```

`\setkeys` This macro sets keys of the form  $\backslash\langle prefix\rangle@\langle family\rangle@\langle key\rangle^3$  where  $\langle family\rangle$  is an element of the list  $\langle families\rangle$  and key is an element of the options list  $\langle keys\rangle$  and not of  $\langle na\rangle$ . The latter list can be used to specify keys that should be ignored by the macro. If a key is defined by more families in the list  $\langle families\rangle$ , the first family from the list defining the key will set it. No errors are produced when  $\langle keys\rangle$  is empty. If  $\langle family\rangle$  is empty, the macro will set keys of the form  $\backslash\langle prefix\rangle@\langle key\rangle$ . If  $\langle prefix\rangle$  is specified and empty, the macro will set keys of the form  $\backslash\langle family\rangle@\langle key\rangle$ . If both  $\langle prefix\rangle$  and  $\langle family\rangle$  are empty, the macro will set keys of the form  $\backslash\langle key\rangle$ .

```
\setkeys [my]{familya,familyb}{keya=test}  
\setkeys [my]{familya,familyb}{keyb=test}  
\setkeys [my]{familyb,familya}{keyb=test}
```

In the example above, line 1 will set keya in family familya. This effectively means that the value test will be submitted to the key macro  $\backslash my@familya@keya$ . The next line will set keyb in familya. The last one sets keyb in family familyb. As the keys used here, directly output their value, the above code results in typesetting the word test three times.

When input is lacking for a key, `\setkeys` will check whether there is a default value for that key that can be used instead. If that is not the case, an error message will be generated. See also section 3.

```
\setkeys [my]{familya}{keya}  
\setkeys [my]{familya}{keyb}
```

The first line of the example above does not generate an error as this key has been defined with a default value (see section 3.1). The second line does generate an error message. See also section 9 for all possible error messages generated by `xkeyval`.

When you want to use commas or equality signs in the value of a key, surround the value by braces, as shown in the example below.

```
\setkeys [my]{familya}{keya={some=text,other=text}}
```

It is possible to nest `\setkeys` commands in other `\setkeys` commands or in key definitions. The following, for instance,

```
\define@key [my]{familyb}{keyc}{#1}  
\setkeys [my]{familyb}{keyc=a\setkeys [my]{familya}{keya=~and b},keyb=~and c}
```

returns a and b and c.

```
\setkeys* [prefix] {families} [na] {keys}
```

`\setkeys*` The starred version of `\setkeys` sets keys which it can locate in the given families and will not produce errors when it cannot find a key. Instead, these keys and their values will be appended to a list of remaining keys in the macro  $\backslash XKV@rm$  after the use of `\setkeys*`. This list will also contain keys and values coming from nested `\setkeys*` commands in the order as they have been submitted. Keys listed in  $\langle na\rangle$  will be ignored fully and will not be appended to the  $\backslash XKV@rm$  list.

```
\setkeys*[my]{familyb}{keya=test}
```

Since keya is not defined in familyb, the value in the example above will be stored in  $\backslash XKV@rm$  (so  $\backslash XKV@rm$  expands to  $keya=test$ ) for later use and no errors are raised.

```
\setrmkeys [prefix] {families} [na]
```

`\setrmkeys` The macro `\setrmkeys` sets the remaining keys given by the list `\XKV@rm` stored previously by a `\setkeys*` (or `\setrmkeys*`) command in *families*. *na* again lists keys that should be ignored. It will produce an error when a key cannot be located.

```
\setrmkeys [my] {familya}
```

This submits `keya=test` from the previous `\setkeys*` command to `familya`. `keya` will be set.

```
\setrmkeys* [prefix] {families} [na]
```

`\setrmkeys*` The macro `\setrmkeys*` acts as the `\setrmkeys` macro but now, as with `\setkeys*`, it ignores keys that it cannot find and puts them again on the list stored in `\XKV@rm`. Keys listed in *na* will be ignored fully and will not be appended to the list in `\XKV@rm`.

```
\setkeys* [my] {familyb} {keya=test}
\setrmkeys* [my] {familyb}
\setrmkeys [my] {familya}
```

In the example above, the second line tries to set `keya` in `familyb` again and no errors are generated on failure. The last line finally sets `keya`.

The combination of `\setkeys*` and `\setrmkeys` can be used to construct complex macros in which, for instance, a part of the keys should be set in multiple families and the rest in another family or set of families. Instead of splitting the keys or the inputs, the user can supply all inputs in a single argument and the two macros will perform the splitting and setting of keys for your macro, given that the families are well chosen.

```
\setkeys+ [prefix] {families} [na] {keys}
\setkeys** [prefix] {families} [na] {keys}
\setrmkeys+ [prefix] {families} [na]
\setrmkeys** [prefix] {families} [na]
```

`\setkeys+`  
`\setkeys**`  
`\setkeys+`  
`\setkeys**` These macros act as their counterparts without the `+`. However, when a key in *keys* is defined by multiple families, this key will be set in *all* families in *families*. This can, for instance, be used to set keys defined by your own package and by another package with the same name but in different families with a single command.

```
\setkeys+ [my] {familya, familyb} {keyb=test}
```

The example above sets `keyb` in both families.

## 4.2 A few details

Several remarks should be made with respect to processing the user input. Assuming that `keya` up to `keyd` are properly defined, one could do the following.

```
\setkeys{family}{keya= test a, keyb={test b,c,d}, , keyc=end}
```

From values consisting entirely of a { } group, the outer braces will be stripped off internally.<sup>8</sup> This allows the user to ‘hide’ any commas or equality signs that appear in the value of a key. This means that when using braces around value, xkeyval will not terminate the value when it encounters a comma in value. For instance, see the value of keyb in the example above. The same holds for the equality sign. Notice further that any spaces around the characters = and , (in the top level group) are removed and that empty entries will silently be ignored. This makes the example above equivalent to the example below.

```
\setkeys{family}{keya=test a,keyb={test b,c,d},keyc=end}
```

Further, when executing a key macro, the following xkeyval internals are available.

`\XKV@prefix`

The prefix, for instance my.

`\XKV@fams`

The list of families to search, for instance familya, familyb.

`\XKV@tfam`

The current family, for instance familya.

`\XKV@header`

The header which is a combination of the prefix and the current family, for instance my@familya@.

`\XKV@tkey`

The current key name, for instance keya.

`\XKV@na`

The keys that should not be set, for instance keyc, keyd.

You can use these internals and create, for example, dynamic options systems in which user input to `\setkeys` will be used to create new keys which can be used in the very same `\setkeys` command. The extract package [1] provides an example for this.

## 5 Pointers

The xkeyval package provides a pointer mechanism. Pointers can be used to copy values of keys. Hence, one can reuse the value that has been submitted to a particular key in the value of another key. This section will first describe how xkeyval can be made to save key values. After that, it will explain how to use these saved values again. Notice already that the commands `\savevalue`, `\gsavevalue` and `\usevalue` can only be used in `\setkeys` commands.

### 5.1 Saving values

`\savevalue` Saving a value for a particular key can be accomplished by using the `\savevalue` command with the key name as argument.

<sup>8</sup>xkeyval actually strips off 3 levels of braces: one by using keyval's `\KV@@sp@def` and two in internal parsings. keyval strips off only 2 levels: one by using `\KV@@sp@def` and one in internal parsings. This difference has not yet been shown to cause problems for existing packages or new implementations. If this appears to be a problem in the future, effort will be done to solve it.

```
\setkeys [my]{familya}{\savevalue{keya}=test}
```

This example will set `keya` as we have seen before, but will additionally define the macro `\XKV@my@familya@keya@value` to expand to `test`. This macro can be used later on by `xkeyval` to replace pointers. In general, values of keys will be stored in macros of the form `\XKV@<prefix>@<family>@<key>@value`. This implies that the pointer system can only be used within the same family (and prefix). We will come back to that in section 5.2.

`\gsavevalue` Using the global version of this command, namely `\gsavevalue`, will define the value macro `\XKV@my@family@key@value` globally. In other words, the value macro won't survive after a `\begingroup . . . \endgroup` construct (for instance, an environment), when it has been created in this group using `\savevalue` and it will survive afterwards if `\gsavevalue` is used.

```
\setkeys [my]{familya}{\gsavevalue{keya}=test}
```

This example will globally define `\XKV@my@familya@keya@value` to expand to `test`.

Actually, in most applications, package authors do not want to require users to use the `\savevalue` form when using the pointer system internally. To avoid this, the `xkeyval` package also supplies the following commands.

```
\savekeys [<prefix>]{<family>}{<keys>}
\gsavekeys [<prefix>]{<family>}{<keys>}
```

`\savekeys` The `\savekeys` macro stores a list of keys for which the values should always be saved to a macro of the form `\XKV@<prefix>@<family>@save`. This will be used by `\setkeys` to check whether a value should be saved or not. The global version will define this internal macro globally so that the settings can escape groups (and environments).  
`\gsavekeys` The `\savekeys` macro works incrementally. This means that new input will be added to an existing list for the family at hand if it is not in yet.

```
\savekeys [my]{familya}{keya,keyc}
\savekeys [my]{familya}{keyb,keyc}
```

The first line stores `keya, keyc` to `\XKV@my@familya@save`. The next line changes the content of this macro to `keya, keyc, keyb`.

```
\delsavekeys [<prefix>]{<family>}{<keys>}
\gdelsavekeys [<prefix>]{<family>}{<keys>}
\unsavekeys [<prefix>]{<family>}
\gunsavekeys [<prefix>]{<family>}
```

`\delsavekeys` The `\delsavekeys` macro can be used to remove some keys from an already defined list of save keys. No errors will be raised when one of the keys in the list `<keys>` was not in the list. The global version `\gdelsavekeys` does the same as `\delsavekeys`, but will also make the resulting list global. The `\unsavekeys` macro can be used to clear the entire list of key names for which the values should be saved. The macro will make `\XKV@<prefix>@<family>@save` undefined. `\gunsavekeys` is similar to `\unsavekeys` but makes the internal macro undefined globally.

```
\savekeys [my]{familya}{keya,keyb,keyc}
\delsavekeys [my]{familya}{keyb}
\unsavekeys [my]{familya}
```

The first line of this example initializes the list to contain `keya`, `keyb`, `keyc`. The second line removes `keyb` from this list and hence `keya`, `keyc` remains. The last line makes the list undefined and hence clears the settings for this family.

`\global`

It is important to notice that the use of the global version `\gsavekeys` will only have effect on the definition of the macro `\XKV@<prefix>@<family>@save`. It will not have an effect on how the key values will actually be saved by `\setkeys`. To achieve that a particular key value will be saved globally (like using `\gsavevalue`), use the `\global` specifier in the `\savekeys` argument.

```
\savekeys [my] {familya} {keya, \global {keyc}}
```

This example does the following. The argument `keya, \global {keyc}` is saved (locally) to `\XKV@my@familya@save`. When `keyc` is used in a `\setkeys` command, the associated value will be saved globally to `\XKV@my@familya@keya@value`. When `keya` is used, its value will be saved locally.

All macros discussed in this section for altering the list of save keys only look at the key name. If that is the same, old content will be overwritten with new content, regardless whether `\global` has been used in the content. See the example below.

```
\savekeys [my] {familya} {\global {keyb}, keyc}
\delsavekeys [my] {familya} {keyb}
```

The first line changes the list in `\XKV@my@familya@save` from `keya, \global {keyc}` to `keya, keyc, \global {keyb}`. The second line changes the list to `keya, keyc`.

## 5.2 Using saved values

`\usevalue`

The syntax of a pointer is `\usevalue{keyname}` and can only be used inside `\setkeys` and friends. `xkeyval` will replace a pointer by the value that has been saved for the key that the pointer is pointing to. If no value has been saved for this key, an error will be raised. The following example will demonstrate how to use pointers (using the keys defined in section 3.1).

```
\setkeys [my] {familya} {\savevalue {keya}=test}
\setkeys [my] {familya} {keyb=\usevalue {keya}}
```

The value submitted to `keyb` points to `keya`. This has the effect that the value recorded for `keya` will replace `\usevalue{keya}` and this value (here `test`) will be submitted to the key macro of `keyb`.

Since the saving of values is prefix and family specific, pointers can only locate values that have been saved for keys with the same prefix and family as the key for which the pointer is used. Hence this

```
\setkeys [my] {familya} {\savevalue {keya}=test}
\setkeys [my] {familyb} {keyb=\usevalue {keya}}
```

will never work. An error will be raised in case a key value points to a key for which the value cannot be found or has not been stored.

It is possible to nest pointers as the next example shows.

```
\setkeys [my] {familya} {\savevalue {keya}=test}
\setkeys [my] {familya} {\savevalue {keyb}=\usevalue {keya}}
\setkeys [my] {familya} {keyc=\usevalue {keyb}}
```

This works as follows. First `xkeyval` records the value `test` in a macro. Then, `keyb` uses that value. Besides that, the value submitted to `keyb`, namely `\usevalue{keya}` will be recorded in another macro. Finally, `keyc` will use the value recorded previously for `keyb`, namely `\usevalue{keya}`. That in turn points to the value saved for `keya` and that value will be used.

It is important to stress that the pointer replacement will be done before  $\TeX$  or  $\LaTeX$  performs the expansion of the key macro and its argument (which is the value that has been submitted to the key). This allows pointers to be used in almost any application. (The exception is grouped material, to which we will come back later.) When programming keys (using `\define@key` and friends), you won't have to worry about the expansion of the pointers which might be submitted to your keys. The value that will be submitted to your key macro in the end, will not contain pointers. These have already been expanded and been replaced by the saved values.

A word of caution is necessary. You might get into an infinite loop if pointers are not applied with care, as the examples below show. The first example shows a direct back link.

```
\setkeys [my]{familya}{\savevalue{keya}=\usevalue{keya}}
```

The second example shows an indirect back link.

```
\setkeys [my]{familya}{\savevalue{keya}=test}
\setkeys [my]{familya}{\savevalue{keyb}=\usevalue{keya}}
\setkeys [my]{familya}{\savevalue{keya}=\usevalue{keyb}}
```

In these cases, an error will be issued and further pointer replacement is canceled.

As mentioned already, pointer replacement does not work inside grouped material, `{ . . . }`, if this group is not around the entire value (since that will be stripped off, see section 1). The following, for instance, will not work.

```
\setkeys [my]{familya}{\savevalue{keya}=test}
\setkeys [my]{familya}{keyb=\parbox{2cm}{\usevalue{keya}}}
```

The following provides a working alternative for this situation.

```
\setkeys [my]{familya}{\savevalue{keya}=test}
\setkeys [my]{familya}{keyb=\begin{minipage}{2cm}\usevalue{keya}\end{minipage}}
```

In case there is no appropriate alternative, we can work around this restriction, for instance by using a value macro directly.

```
\setkeys [my]{familya}{\savevalue{keya}=test}
\setkeys [my]{familya}{keyb=\parbox{2cm}{\XKV@my@familya@keya@value}}
```

When no value has been saved for `keya`, the macro `\XKV@my@familya@keya@value` is undefined. Hence one might want to do a preliminary check to be sure that the macro exists.

Pointers can also be used in default values. We finish this section with an example which demonstrates this.

```
\define@key {fam}{keya}{keya: #1}
\define@key {fam}{keyb}{\usevalue{keya}}{keyb: #1}
\define@key {fam}{keyc}{\usevalue{keyb}}{keyc: #1}
\setkeys {fam}{\savevalue{keya}=test}
\setkeys {fam}{\savevalue{keyb}}
\setkeys {fam}{keyc}
```

Since user input is lacking in the final two commands, the default value defined for those keys will be used. In the first case, the default value points to `keya`, which results in the value `test`. In the second case, the pointer points to `keyb`, which points to `keya` (since its value has been saved now) and hence also in the final command, the value `test` will be submitted to the `key` macro of `keyc`.

## 6 Presetting keys

In contrast to the default value system where users are required to specify the key without a value to assign it its default value, the presetting system does not require this. Keys which are preset will be set automatically by `\setkeys` when the user didn't use those keys in the `\setkeys` command. When users did use the keys which are also preset, `\setkeys` will avoid setting them again. This section again uses the key definitions of section 3.1 in examples.

```
\presetkeys [prefix] {family}{head keys}{tail keys}
\gpresetkeys [prefix] {family}{head keys}{tail keys}
```

`\presetkeys`  
`\gpresetkeys`

This macro will save *head keys* to `\XKV@prefix@family@preseth` and *tail keys* to `\XKV@prefix@family@presett`. Savings are done locally by `\presetkeys` and globally by `\gpresetkeys` (compare `\savekeys` and `\gsavekeys`, section 5.1). The saved macros will be used by `\setkeys`, when they are defined, whenever *family* is used in the *families* argument of `\setkeys`. Head keys will be set before setting user keys, tail keys will be set afterwards. However, if a key appears in the user input, this particular key will not be set by any of the preset keys.

The macros work incrementally. This means that new input for a particular key replaces already present settings for this key. If no settings were present yet, the new input for this key will be appended to the end of the existing list. The replacement ignores the fact whether a `\savevalue` or an `=` has been specified in the key input. We could do the following.

```
\presetkeys{fam}{keya=red,\savevalue{keyb},keyc}{ }
\presetkeys{fam}{\savevalue{keya},keyb=red,keyd}{ }
```

After the first line of the example, the macro `\XKV@KV@fam@preseth` will contain `keya=red,\savevalue{keyb},keyc`. After the second line of the example, the macro will contain `\savevalue{keya},keyb=red,keyc,keyd`. The *tail keys* remain empty throughout the example.

```
\delpresetkeys [prefix] {family}{head keys}{tail keys}
\gdelpresetkeys [prefix] {family}{head keys}{tail keys}
```

`\delpresetkeys`  
`\gdelpresetkeys`

These commands can be used to (globally) delete entries from the presets by specifying the key names for which the presets should be deleted. Continuing the previous example, we could do the following.

```
\delpresetkeys{fam}{keya,keyb}{ }
```

This redefines the list of head presets `\XKV@KV@fam@preseth` to contain `keyc,keyd`. As can be seen from this example, the exact use of a key name is irrelevant for successful deletion.

```
\unpresetkeys [<prefix>]{<family>}
\gunpresetkeys [<prefix>]{<family>}
```

\unpresetkeys  
\gunpresetkeys

These commands clear the presets for *<family>* and works just as `\unsavekeys`. It makes `\XKV@<prefix>@<family>@preseth` and `\XKV@<prefix>@<family>@presett` undefined. The global version will make the macros undefined globally.

Two type of problems in relation to pointers could appear when specifying head and tail keys incorrectly. This will be demonstrated with two examples. In the first example, we would like to set `keya` to `blue` and `keyb` to copy the value of `keya`, also when the user has changed the preset value of `keya`. Say that we implement the following.

```
\savekeys [my] {familya}{keya}
\presetkeys [my] {familya}{keya=blue, keyb=\usevalue{keya}}{}
\setkeys [my] {familya}{keya=red}
```

This will come down to executing

```
\savekeys [my] {familya}{keya}
\setkeys [my] {familya}{keyb=\usevalue{keya}, keya=red}
```

since `keya` has been specified by the user. At best, `keyb` will copy a probably wrong value of `keya`. In the case that no value for `keya` has been saved before, we get an error. We observe that the order of keys in the simplified `\setkeys` command is wrong. This example shows that the `keyb=\usevalue{keya}` should have been in the tail keys, so that it can copy the user input to `keya`.

The following example shows what can go wrong when using presets incorrectly and when `\setkeys` contains pointers.

```
\savekeys [my] {familya}{keya}
\presetkeys [my] {familya}{}{keya=red}
\setkeys [my] {familya}{keyb=\usevalue{keya}}
```

This will come down to executing the following.

```
\savekeys [my] {familya}{keya}
\setkeys [my] {familya}{keyb=\usevalue{keya}, keya=red}
```

This results in exactly the same situation as we have seen in the previous example and hence the same conclusion holds. In this case, we conclude that the `keya=red` argument should have been specified in the head keys of the `\presetkeys` command so that `keyb` can copy the value of `keya`.

For most applications, one could use the rule of thumb that preset keys containing pointers should go in the tail keys. All other keys should go in head keys. There might, however, be applications thinkable in which one would like to implement the preset system as shown in the two examples above, for instance to easily retrieve values used in the last use of a macro or environment. However, make sure that keys in that case receive an initialization in order to avoid errors of missing values.

For completeness, the working examples are below.

```
\savekeys [my] {familya}{keya}
\presetkeys [my] {familya}{keya=blue}{keyb=\usevalue{keya}}
\setkeys [my] {familya}{keya=red}
\presetkeys [my] {familya}{keya=red}{}
\setkeys [my] {familya}{keyb=\usevalue{keya}}
```

Other examples can be found in the example files. See section 13.



## 7 Package option processing

The macros in this section can be used to build  $\LaTeX$  class or package options systems using `xkeyval`. These are comparable to the standard  $\LaTeX$  macros without the trailing `X`. See for more information about these  $\LaTeX$  macros the documentation of the source [2] or a  $\LaTeX$  manual (for instance, the  $\LaTeX$  Companion [4]). The macros in this section have been built using `\define@key` and `\setkeys` and are not available to  $\TeX$  users.

The macros below allow for specifying the  $\langle family \rangle$  (or  $\langle families \rangle$ ) as an optional argument. This could be useful if you want to define global options which can be reused later (and set locally by the user) in a macro or environment that you define. If no  $\langle family \rangle$  (or  $\langle families \rangle$ ) is specified, the macro will insert the default family name which is the filename of the file that is calling the macros. The macros in this section also allow for setting an optional prefix. When using the filename as family, uniqueness of key macros is already guaranteed. In that case, you can omit the optional  $\langle prefix \rangle$ . However, when you use a custom prefix for other keys in your package and you want to be able to set all of the keys later with a single command, you can use the custom prefix also for the class or package options system.

Note that both  $[\langle arg \rangle]$  and  $\langle \langle arg \rangle \rangle$  denote optional arguments to the macros in this section. This syntax is used to identify the different optional arguments when they appear next to each other.

```
\DeclareOptionX[ $\langle prefix \rangle$ ] $\langle family \rangle$ >{ $\langle key \rangle$ }[ $\langle default \rangle$ ]{ $\langle function \rangle$ }
```

`\DeclareOptionX` Declares an option (i.e., a key, which can also be used later on in the package in `\setkeys` and `friends`). This macro is comparable to the standard  $\LaTeX$  macro `\DeclareOption`, but with this command, the user can pass a value to the option as well. Reading that value can be done by using `#1` in  $\langle function \rangle$ . This will contain  $\langle default \rangle$  when no value has been specified for the key. The value of the optional argument  $\langle default \rangle$  is empty by default. This implies that when the user does not assign a value to  $\langle key \rangle$  and when no default value has been defined, no error will be produced. The optional argument  $\langle family \rangle$  can be used to specify a custom family for the key. When the argument is not used, the macro will insert the default family name.

```
\newif\iflandscape
\DeclareOptionX{landscape}{\landscapetrue}
\DeclareOptionX{parindent}[20pt]{\setlength\parindent{#1}}
```

Assuming that the file containing the example above is called `myclass.cls`, the example is equivalent to

```
\newif\iflandscape
\define@key{myclass.cls}{landscape}[]{\landscapetrue}
\define@key{myclass.cls}{parindent}[20pt]{\setlength\parindent{#1}}
```

Notice that an empty default value has been inserted by `xkeyval` for the `landscape` option. This allows for the usual  $\LaTeX$  options use like

```
\documentclass[landscape]{myclass}
```

without raising `No value specified for key 'landscape'` errors.

These examples also show that one can also use `\define@key` (or `friends`, see section 3) to define class or package options. The macros presented here are supplied for the ease of package programmers wanting to convert the options section of their package to use `xkeyval`.

```
\DeclareOptionX*{<function>}
```

`\DeclareOptionX*` This macro can be used to process any unknown inputs. It is comparable to the  $\LaTeX$  macro `\DeclareOption*`. Use `\CurrentOption` within this macro to get the entire input from which the key is unknown, for instance `unknownkey=value` or `somevalue`. These values (possibly including a key) could for example be passed on to another class or package or could be used as an extra class or package option specifying for instance a style that should be loaded.

```
\DeclareOptionX*{\PackageWarning{mypackage}{'\CurrentOption' ignored}}
```

The example produces a warning when the user issues an option that has not been declared.

```
\ExecuteOptionsX[<prefix>]<families>[<na>]{<keys>}
```

`\ExecuteOptionsX` This macro sets keys created by `\DeclareOptionX` and is basically a copy of `\setkeys`. The optional argument `<na>` specifies keys that should be ignored. The optional argument `<families>` can be used to specify a list of families which define `<keys>`. When the argument is not used, the macro will insert the default family name. This macro will not use the declaration done by `\DeclareOptionX*` when undeclared options appear in its argument. Instead, in this case the macro will raise an error. This mimics  $\LaTeX$ 's `\ExecuteOptions`' behavior.

```
\ExecuteOptionsX{parindent=0pt}
```

This initializes `\parindent` to `0pt`.

```
\ProcessOptionsX[<prefix>]<families>[<na>]
```

`\ProcessOptionsX` This macro processes the keys and values passed by the user to the class or package. The optional argument `<na>` can be used to specify keys that should be ignored. The optional argument `<families>` can be used to specify the families that have been used to define the keys. Note that this macro will not protect macros in the user inputs (like `\thepage`) as will be explained in section 12.2. When used in a class file, this macro will ignore unknown keys or options. This allows the user to use global options in the `\documentclass` command which can be copied by packages loaded afterwards.

```
\ProcessOptionsX* [ <prefix> ] <families> > [ <na> ]
```

`\ProcessOptionsX*` The starred version works like `\ProcessOptionsX` except that it also copies user input from the `\documentclass` command. When the user specifies an option in the document class which also exists in the local family (or families) of the package issuing `\ProcessOptionsX*`, the local key will be set as well. In this case, #1 in the `\DeclareOptionX` macro will contain the value entered in the `\documentclass` command for this key. First the global options from `\documentclass` will set local keys and afterwards, the local options, specified with `\usepackage`, `\RequirePackage` or `\LoadClass` (or friends), will set local keys, which could overwrite the global options again, depending on the way the options sections are constructed. This macro reduces to `\ProcessOptionsX` only when issued from the class which forms the document class for the file at hand to avoid setting the same options twice, but not for classes loaded later using for instance `\LoadClass`. Global options that do not have a counterpart in local families of a package or class will be skipped.

It should be noted that these implementations differ from the  $\LaTeX$  implementations of `\ProcessOptions` and `\ProcessOptions*`. The difference is in copying the global options. The  $\LaTeX$  commands always copy global options if possible. As a package author doesn't know beforehand which document class will be used and with which options, the options declared by the author might show some unwanted interactions with the global options. When the class and the package share the same option, specifying this option in the `\documentclass` command will force the package to use that option as well. With `\ProcessOptionsX`, `xkeyval` offers a package author to become fully independent of the global options and be sure to avoid conflicts with any class. Have a look at the example class, style and `.tex` file below and observe the effect of changing to `\ProcessOptionsX*` in the style file.<sup>9</sup>

```
% myclass.cls
\RequirePackage{xkeyval}
\define@boolkey{myclass.cls}%
  [cls]{bool}{}
\ProcessOptionsX
\LoadClass{article}
```

```
% mypack.sty
\define@boolkey{mypack.sty}%
  [pkg]{bool}{}
\ProcessOptionsX
```

```
% test.tex
\documentclass[bool=true]{myclass}
\usepackage{mypack}
\begin{document}\parindent0pt
\ifclsbool class boolean true \else class boolean false\fi\
\ifpkgbool package boolean true \else package boolean false\fi
\end{document}
```

See section 13 for more examples.

The use of `\ProcessOptionsX*` in a class file might be tricky since the class could also be used as a basis for another package or class using `\LoadClass`. In that case, depending on the options system of the document class, the behavior of the class loaded with `\LoadClass` could change compared to the situation when it is loaded by `\documentclass`. But since it is technically possible to create two classes that cooperate, the `xkeyval` package allows for the usage of `\ProcessOptionsX*` in class files. Notice that using  $\LaTeX$ 's `\ProcessOptions` or `\ProcessOptions*`, a class file cannot copy document class options.

In case you want to verify whether your class is loaded with `\documentclass` or `\LoadClass`, you can use the `\XKV@documentclass` macro which contains the current document class.

A final remark concerns the use of expandable macros in class or package options values. Due to the construction of the  $\LaTeX$  option processing mechanism, this is not possible. However, the `xkeyval` bundle includes a patch for the  $\LaTeX$  kernel which solves this problem. See section 14.6 for more information.

## 8 List of macro structures

This section provides a list of all reserved internal macro structures used for key processing. Here `pre` denotes a prefix, `fam` denotes a family and `key` denotes a key. These vary per application. The other parts in internal macro names are constant. The macros with additional `XKV` prefix are protected in the sense that all `xkeyval` macros

<sup>9</sup>See section 3.4 for information about `\define@boolkey`.

disallow the use of the XKV prefix. Package authors using `xkeyval` are responsible for protecting the other types of internal macros.

`\pre@fam@key`

Key macro. This macro takes one argument. This macro will execute the *<function>* of `\define@key` (and friends) on the value submitted to the key macro through `\setkeys`.

`\cmdpre@fam@key`

The macro which is used by `\define@cmdkey` to store user input in when no custom macro prefix was specified.

`\ifpre@fam@key`, `\pre@fam@keytrue`, `\pre@fam@keyfalse`

The conditional created by `\define@boolkey` with parameters `pre`, `fam` and `key` if no custom macro prefix was specified. The `true` and `false` macros are used to set the conditional to `\iftrue` and `\iffalse` respectively.

`\pre@fam@key@default`

Default value macro. This macro expands to `\pre@fam@key{default value}`. This macro is defined through `\define@key` and friends.

`\XKV@pre@fam@key@value`

This macro is used to store the value that has been submitted through `\setkeys` to the key macro (without replacing pointers).

`\XKV@pre@fam@save`

Contains the names of the keys that should always be saved when they appear in a `\setkeys` command. This macro is defined by `\savekeys`.

`\XKV@pre@fam@preseth`

Contains the head presets. These will be submitted to `\setkeys` before setting user input. Defined by `\presetkeys`.

`\XKV@pre@fam@presett`

Contains the tail presets. These will be submitted to `\setkeys` after setting user input. Defined by `\presetkeys`.

An important remark should be made. Most of the macros listed above will be constructed by `xkeyval` internally using `\csname . . . \endcsname`. Hence almost any input to the macros defined by this package is possible. However, some internal macros might be used outside `xkeyval` macros as well, for instance the macros of the form `\ifpre@fam@key` and `\cmdpre@fam@key`. To be able to use these macros yourself, none of the input parameters should contain non-letter characters. If you feel that this is somehow necessary anyway, there are several strategies to make things work.

Let us consider as example the following situation (notice the hyphen - in the family name).

```
\define@boolkey{some-fam}{myif}  
\setkeys{some-fam}{myif=false}
```

Using these keys in a `\setkeys` command is not a problem at all. However, if you want to use the `\ifKV@some-fam@myif` command itself, you can do either

```

\edef\savedhyphencatcode{\the\catcode'\-}%
\catcode'\-=11\relax
\def\mymacro{%
  \ifKV@some-fam@myif
    % true case
  \else
    % false case
  \fi}
\catcode'\-=\savedhyphencatcode

```

or

```

\def\mymacro{%
  \csname ifKV@some-fam@myif\endcsname
  % true case
\else
  % false case
\fi}

```

## 9 Warnings and errors

There are several points where `xkeyval` performs a check and could produce a warning or an error. All possible warnings or and error messages are listed below with an explanation. Here `pre` denotes a prefix, `name` denotes the name of a key, `fam` denotes a family, `fams` denotes a list of families and `val` denotes some value. These vary per application. Note that messages 1 to 7 could result from erroneous key setting through `\setkeys`, `\setrmkeys`, `\ExecuteOptionsX` and `\ProcessOptionsX`.

- 1) value ‘`val`’ is not allowed (error)  
The value that has been submitted to a key macro is not allowed. This error can be generated by either a choice or a boolean key.
- 2) ‘`name`’ undefined in families ‘`fams`’ (error)  
The key name is not defined in the families in `fams`. Probably you mistyped name.
- 3) no key specified for value ‘`val`’ (error)  
`xkeyval` found a value without a key, for instance something like `=value`, when setting keys.
- 4) no value recorded for key ‘`name`’ (error)  
You have used a pointer to a key for which no value has been saved previously.
- 5) back linking pointers; pointer replacement canceled (error)  
You were back linking pointers. Further pointer replacements are canceled to avoid getting into an infinite loop. See section 5.2.
- 6) no value specified for key ‘`name`’ (error)  
You have used the key ‘`name`’ without specifying any value for it (namely, `\setkeys{fam}{name}` and the key does not have a default value. Notice that `\setkeys{fam}{name=}` submits the empty value to the key macro and that this is considered a legal value.
- 7) key ‘`name`’ has been disabled (warning)  
The key that you try to set has been disabled and cannot be used anymore.

- 8) ‘XKV’ prefix is not allowed (error)  
 You were trying to use the XKV prefix when defining or setting keys. This error can be caused by any xkeyval macro having an optional prefix argument.
- 9) key ‘name’ undefined (error)  
 This error message is caused by trying to disable a key that does not exist. See section 3.6.
- 10) no save keys defined for ‘pre@fam@’ (error)  
 You are trying to delete or undefine save keys that have not been defined yet. See section 5.1.
- 11) no presets defined for ‘pre@fam@’ (error)  
 You are trying to delete or undefine presets that have not been defined yet. See section 6.
- 12) xkeyval loaded before \documentclass (error)  
 Load xkeyval after \documentclass (or in the class that is the document class). See section 7.

## 10 Category codes

Some packages change the catcode of the equality sign and the comma. This is a problem for keyval as it then does not recognize these characters anymore and cannot parse the input. This problem can play up on the background. Consider for instance the following example and note that the graphicx package is using keyval and that Turkish babel will activate the equality sign for shorthand notation.

```
\documentclass{article}
\usepackage{graphicx}
\usepackage[turkish]{babel}
\begin{document}
\includegraphics[scale=.5]{rose.eps}
\end{document}
```

The babel package provides syntax to temporarily reset the catcode of the equality sign and switch shorthand back on after using keyval (in the `\includegraphics` command), namely `\shorthandoff{=}` and `\shorthandon{=}`. But having to do this every time keyval is invoked is quite cumbersome. Besides that, it might not always be clear to the user what the problem is and what the solution.

For these reasons, xkeyval performs several actions with user input before trying to parse it.<sup>10</sup> First of all, it performs a check whether the characters = and , appear in the input with unexpected catcodes. If so, the `\@selective@sanitize` macro is used to sanitize these characters only in the top level. This means that characters inside (a) group(s), { }, will not be sanitized. For instance, when using Turkish babel, it is possible to use = shorthand notation even in the value of a key, as long as this value is inside a group.

```
\documentclass{article}
\usepackage{graphicx}
```

<sup>10</sup>Notice that temporarily resetting catcodes before reading the input to `\setkeys` won't suffice, as it will not help solving problems when input has been read before and has been stored in a token register or a macro.

```

\usepackage[turkish]{babel}
\usepackage{xkeyval}
\makeatletter
\define@key{fam}{key}{#1}
\begin{document}
\includegraphics[scale=.5]{rose.eps}
\setkeys{fam}{key={some =text}}
\end{document}

```

In the example above, the `\includegraphics` command does work. Further, the first equality sign in the `\setkeys` command will be sanitized, but the second one will be left untouched and will be typeset as babel shorthand notation.

The commands `\savekeys` and `\disable@keys` are protected against catcode changes of the comma. The commands `\setkeys` and `\presetkeys` are protected against catcode changes of the comma and the equality sign. Note that  $\TeX$  option macros (see section 7) are not protected as  $\TeX$  does not protect them either.

## 11 Known issues

This package redefines `keyval`'s `\define@key` and `\setkeys`. This is risky in general. However, since `xkeyval` extends the possibilities of these commands while still allowing for the `keyval` syntax and use, there should be no problems for packages using these commands after loading `xkeyval`. The package prevents `keyval` to be loaded afterwards to avoid these commands from being redefined again into the simpler versions. For packages using internals of `keyval`, like `\KV@sp@def`, `\KV@do` and `\KV@errx`, these are provided separately in `keyval.tex`.

The advantage of redefining these commands instead of making new commands is that it is much easier for package authors to start using `xkeyval` instead of `keyval`. Further, it eliminates the confusion of having multiple commands doing similar things.

A potential problem lies in other packages that redefine either `\define@key` or `\setkeys` or both. Hence particular care has been spend to check packages for this. Only one package has been found to do this, namely `pst-key`. This package implements a custom version of `\setkeys` which is specialized to set PSTricks [5, 6] keys of the form `\psset@somekey`. `xkeyval` also provides the means to set these kind of keys (see page 4) and work is going on to convert PSTricks packages to be using a specialization of `xkeyval` instead of `pst-key`. This specialization is available in the `pst-xkey` package, which is distributed with the `xkeyval` bundle and is described in section 12.3. However, since a lot of authors are involved and since it requires a change of policy, the conversion of PSTricks packages might take some time. Hence, at the moment of writing, `xkeyval` will conflict with `pst-key` and the PSTricks packages still using `pst-key`, which are `pst-ob3d`, `pst-stru` and `pst-uml`.

Have a look at the PSTricks website [5] to find out if the package that you want to use has been converted already. If not, load an already converted package (like `psstricks-add`) after loading the old package to make them work.

## 12 Additional packages

### 12.1 xkvview

The `xkeyval` bundle includes a viewer utility, called `xkvview`,<sup>11</sup> which keeps track of the keys that are defined. This utility is intended for package programmers who want to have an overview of the keys defined in their package(s). All keys defined after loading the package will be recorded in a database. It provides the following commands to display (part of) the database.

```
\xkvview{<options>}
```

`\xkvview` When *<options>* is empty, the entire database will be typeset in a table created with the `longtable` package. The columns will, respectively, contain the key name, the prefix, the family, the type (ordinary, command, choice or boolean) and the presence of a default value for every key defined after loading `xkvview`.

*options* There are several options to control the output of this command. This set of options can be used to set up criteria for the keys that should be displayed. If a key does not satisfy one or more of them, it won't be included in the table. For instance, the following example will display all keys with family `fama`, that do not have a default value. Notice that `xkvview` codes 'no default value' with `[none]`.

*prefix*  
*family*  
*type*  
*default*

```
\documentclass{article}
\usepackage{xkvview}
\makeatletter
\define@key{fama}{keya}[default]{}
\define@cmdkey{fama}{keyb}{}
\define@choicekey{famb}{keyc}{a,b}{}
\define@boolkey{famb}{keyd}{}
\makeatother
\begin{document}
\xkvview{family=fama,default=[none]}
\end{document}
```

In the following examples in this section, the same preamble will be used, but will not be displayed explicitly in the examples.

*option* One can select the columns that should be included in the table using the `columns` option. The following example includes the columns `prefix` and `family` in the table (additional to the key name column).

*columns*

```
\xkvview{columns={prefix,family}}
```

The remaining columns are called `type` and `default`.

*option* If you want to refer to an option, `\xkvview` can automatically generate labels using the scheme *<prefix>-<family>-<keyname>*. Here is an example.

*vlabels*

```
\xkvview{vlabels=true}
Find more information about the keya
option on page~\pageref{KV-fama-keya}.
```

*options* The package can also write (part of) the database to a file. The selection of the information happens in the same way as discussed above. When specifying a filename with the option `file`, the body of the table that is displayed, will also be written to this file. Entries will be separated by `wcolsep` which is `&` by default and every row

*file*  
*wcolsep*  
*weol*

<sup>11</sup>The `xkvview` package is contained in the file `xkvview.sty`.



will be concluded by `weol` which is `\\` by default. The output in the file can then be used as basis for a custom table, for instance in package documentation. The following displays a table in the dvi and also writes the body to `out.tex`.

```
\xkvview{file=out}
```

`out.tex` contains

```
keya&KV&fama&ordinary&default\\
keyb&KV&fama&command&[none]\\
keyc&KV&famb&choice&[none]\\
keyd&KV&famb&boolean&[none]\\
```

The following example generates a table with entries separated by a space and no end-of-line content.

```
\xkvview{file=out,wcolsep=\space,weol=}
```

Now `out.tex` contains

```
keya KV fama ordinary default
keyb KV fama command [none]
keyc KV famb choice [none]
keyd KV famb boolean [none]
```

*option*  
*wlabels*

When post-processing the table generated in this way, one might want to refer to entries again as well. When setting `wlabels` to true, the labels with names  $\langle prefix \rangle - \langle family \rangle - \langle keyname \rangle$  will be in the output file. The following

```
\xkvview{file=out,wlabels=true}
```

will result in the following content in `out.tex`

```
keya&KV&fama&ordinary&default\label{KV-fama-keya}\\
keyb&KV&fama&command&[none]\label{KV-fama-keyb}\\
keyc&KV&famb&choice&[none]\label{KV-famb-keyc}\\
keyd&KV&famb&boolean&[none]\label{KV-famb-keyd}\\
```

*option*  
*view*

Finally, when you only want to generate a file and no output to the dvi, set the `view` option to false.

```
\xkvview{file=out,view=false}
```

This example only generate `out.tex` and does not put a table in the dvi.

## 12.2 xkvltxp

The package and class option system of  $\text{\LaTeX}$  contained in the kernel performs some expansions while processing options. This prevents doing for instance

```
\documentclass[title=My title,author=\textsc{Me}]{myclass}
```

given that `myclass` uses `xkeyval` and defines the options `title` and `author`.

This problem can be overcome by redefining certain kernel commands. These redefinitions are contained in the `xkvltxp` package.<sup>12</sup> If you want to allow the user of your class to be able to specify expandable macros in the package options, the user will have

<sup>12</sup>The `xkvltxp` package consists of the file `xkvltxp.sty`.

to do `\RequirePackage{xkvltxp}` on the first line of the  $\TeX$  file. If you want to offer this functionality in a package, the user can use the package in the ordinary way with `\usepackage{xkvltxp}`. This package then has to be loaded before loading the package which will use this functionality. A description of the patch can be found in the source code documentation.

The examples below summarize this information. The first example shows the case in which we want to allow for macros in the `\documentclass` command.

```
\RequirePackage{xkvltxp}
\documentclass[title=My title,author=\textsc{Me}]{myclass}
\begin{document}
\end{document}
```

The second example shows the case in which we want to allow for macros in a `\usepackage` command.

```
\documentclass{article}
\usepackage{xkvltxp}
\usepackage[footer=page~\thepage.]{mypack}
\begin{document}
\end{document}
```

Any package or class using `xkeyval` and `xkvltxp` to process options can take options that contain macros in their value without expanding them prematurely. However, you can of course not use macros in options which are not of the `key=value` form since they might in the end be passed on to or copied by a package which is not using `xkeyval` to process options, which will then produce errors. Options of the `key=value` form will be deleted from `\@classoptionslist` (see section 7) and form no threat for packages loaded later on. Finally, make sure not to pass options of the `key=value` form to packages not using `xkeyval` to process options since they cannot process them. For examples see section 13.

### 12.3 pst-xkey

The `pst-xkey` package<sup>13</sup> implements a specialized version of the options system of `xkeyval` designed for `PSTricks` [5, 6]. This system gives additional freedom to `PSTricks` package authors since they won't have to worry anymore about potentially redefining keys of one of the many other `PSTricks` packages. The command `\psset` is redefined to set keys in multiple families. Reading the documentation of the `xkeyval` package (especially section 11) first is recommended.

Keys defined in the original distribution of `PSTricks` have the macro structure `\psset@somekey` (where `psset` is literal). These can be (re)defined by

```
\define@key [psset] {} {somekey} {function}
```

Notice especially that these keys are located in the so-called 'empty family'. For more information about `\define@key` and friends, see section 3.

When writing a `PSTricks` package, let's say `pst-new`, you should locate keys in a family which contains the name of your package. If you only need one family, you should define keys using

```
\define@key [psset] {pst-new} {somekey} {function}
```

<sup>13</sup>The `pst-xkey` package consists of the files `pst-xkey.tex` and `pst-xkey.sty`. To load `pst-xkey`  $\TeX$  users do `\input pst-xkey`,  $\LaTeX$  users do `\RequirePackage{pst-xkey}` or `\usepackage{pst-xkey}`.

If you want to use multiple families in your package, you can do

```
\define@key [psset] {pst-new-a} {somekey} {function}
\define@key [psset] {pst-new-b} {anotherkey} {function}
```

`\pst@addfams`  
`\pst@famlist`

It is important that you add all of the families that you use in your package to the list in `\pst@famlist`. This list of families will be used by `\psset` to scan for keys to set user input. You can add your families to the list using

```
\pst@addfams {<families>}
```

For instance

```
\pst@addfams {pst-new}
```

or

```
\pst@addfams {pst-new-a, pst-new-b}
```

Only one command is needed to set PSTricks keys.

```
\psset [<families>] {<keys>}
```

`\psset`

This command will set `<keys>` in `<families>` using `\setkeys+` (see section 4). When `<families>` is not specified, it will set `<keys>` in all families in `\pst@famlist` (which includes the empty family for original PSTricks keys).

```
\psset {somekey=red, anotherkey}
\psset [pst-new-b] {anotherkey=green}
```

## 13 Examples and documentation

To generate the package and example files from the source, find the source of this package, the file `xkeyval.dtx`, in your local  $\TeX$  installation or on CTAN and run it with  $\LaTeX$ .

```
latex xkeyval.dtx
```

This will generate the package files (`xkeyval.tex`, `xkeyval.sty`, `xkvltxp.sty`, `keyval.tex`, `xkvtxhdr.tex`, `xkvview.sty`, `pst-xkey.tex` and `pst-xkey.sty`) and the example files.

The file `xkvex1.tex` provides an example for  $\TeX$  users for the macros described in sections 3, 4, 5 and 6. The file `xkvex2.tex` provides an example for  $\LaTeX$  users for the same macros. The files `xkvex3.tex`, `xkveca.cls`, `xkvecb.cls`, `xkvesa.sty`, `xkvesb.sty` and `xkvesc.sty` together form an example for the macros described in section 7. The set of files `xkvex4.tex`, `xkveca.cls`, `xkvecb.cls`, `xkvesa.sty`, `xkvesb.sty` and `xkvesc.sty` provides an example for sections 7 and 12.2. These files also demonstrate the possibilities of interaction between packages or classes not using `xkeyval` and packages or classes that do use `xkeyval` to set options.

To (re)generate this documentation, perform the following steps.

```
latex xkeyval.dtx
latex xkeyval.dtx
bibtex xkeyval
makeindex -s gglo.ist -o xkeyval.gls xkeyval.glo
makeindex -s gind.ist -o xkeyval.ind xkeyval.idx
```

```
latex xkeyval.dtx
latex xkeyval.dtx
```

## 14 Implementation

### 14.1 xkeyval.tex

Avoid loading xkeyval.tex twice.

```
1 (*xkv.tex)
2 \csname XKeyValLoaded\endcsname
3 \let\XKeyValLoaded\endinput
```

Adjust some catcodes to define internal macros.

```
4 \edef\XKVcatcodes{%
5   \catcode'\noexpand\@the\catcode'\@relax
6   \catcode'\noexpand\=\the\catcode'\relax
7   \catcode'\noexpand\,\the\catcode'\,\relax
8   \catcode'\noexpand\:\the\catcode'\:\relax
9   \let\noexpand\XKVcatcodes\relax
10 }
11 \catcode'\@11\relax
12 \catcode'\=12\relax
13 \catcode'\,12\relax
14 \catcode'\:12\relax
```

Initializations. This package uses a private token to avoid conflicts with other packages that use  $\LaTeX$  scratch token registers in key macro definitions (for instance, graphicx, keys angle and scale).

```
15 \newtoks\XKV@toks
16 \newtoks\XKV@tempa@toks
17 \newcount\XKV@depth
18 \newif\ifXKV@st
19 \newif\ifXKV@sg
20 \newif\ifXKV@pl
21 \newif\ifXKV@knf
22 \newif\ifXKV@rkv
23 \newif\ifXKV@inpox
24 \newif\ifXKV@preset
25 \let\XKV@rm\@empty
```

Load  $\LaTeX$  primitives if necessary and provide information.

```
26 \ifx\ProvidesFile\@undefined
27   \message{2008/08/13 v2.6a key=value parser (HA)}
28   \input xkvtxhdr
29 \else
30   \ProvidesFile{xkeyval.tex}[2008/08/13 v2.6a key=value parser (HA)]
31   \@addtofilelist{xkeyval.tex}
32 \fi
```

`\@firstoftwo` Two utility macros from the latex.ltx needed for executing `\XKV@ifundefined` in the sequel.

```
33 \long\def\@firstoftwo#1#2{#1}
34 \long\def\@secondoftwo#1#2{#2}
```

`\XKV@afterfi` Two utility macros to move execution of content of a conditional branch after the `\fi`.  
`\XKV@afterelsefi` This avoids nesting conditional structures too deep.

```

35 \long\def\XKV@afterfi#1\fi{\fi#1}
36 \long\def\XKV@afterelsefi#1\else#2\fi{\fi#1}

```

`\XKV@ifundefined` `{\langle csname \rangle}{\langle undefined \rangle}{\langle defined \rangle}`  
Executes `\langle undefined \rangle` if the control sequence with name `\langle csname \rangle` is undefined, else it executes `\langle defined \rangle`. This macro uses  $\epsilon$ -TeX if possible to avoid filling TeX's hash when checking control sequences like key macros in the rest of the package. The use of `\XKV@afterelsefi` is necessary here to avoid TeX picking up the second `\fi` as end of the main conditional when `\ifcsname` is undefined. For `\XKV@afterelsefi` this `\fi` is hidden in the group used to define `\XKV@ifundefined` in branch of the case that `\ifcsname` is defined. Notice the following. Both versions of the macro leave the tested control sequence undefined. However, the first version will execute `\langle undefined \rangle` if the control sequence is undefined or `\relax`, whereas the second version will only execute `\langle undefined \rangle` if the control sequence is undefined. This is no problem for the applications in this package.

```

37 \ifx\ifcsname\@undefined\XKV@afterelsefi
38   \def\XKV@ifundefined#1{%
39     \begingroup\expandafter\expandafter\expandafter\endgroup
40     \expandafter\ifx\csname#1\endcsname\relax
41     \expandafter\@firstoftwo
42   \else
43     \expandafter\@secondoftwo
44   \fi
45 }
46 \else
47   \def\XKV@ifundefined#1{%
48     \ifcsname#1\endcsname
49     \expandafter\@secondoftwo
50   \else
51     \expandafter\@firstoftwo
52   \fi
53 }
54 \fi

```

Check whether `keyval` has been loaded and if not, load `keyval` primitives and prevent `keyval` from being loaded after `xkeyval`.

```

55 \XKV@ifundefined{ver@keyval.sty}{
56   \input keyval
57   \expandafter\def\csname ver@keyval.sty\endcsname{1999/03/16}
58 }{}

```

`\@ifnextcharacter` Check the next character independently of its catcode. This will be used to safely perform `\@ifnextcharacter+` and `\@ifnextcharacter*`. This avoids errors in case any other package changes the catcode of these characters.

`\@ifncharacter`

Contributed by Donald Arseneau.

```

59 \long\def\@ifnextcharacter#1#2#3{%
60   \@ifnextchar\bgroup
61   {\@ifnextchar{#1}{#2}{#3}}%
62   {\@ifncharacter{#1}{#2}{#3}}%
63 }

```

```

64 \long\def@ifncharacter#1#2#3#4{%
65   \if\string#1\string#4%
66     \expandafter\@firstoftwo
67   \else
68     \expandafter\@secondoftwo
69   \fi
70   {#2}{#3}#4%
71 }

\XKV@for@n  {<list>}<cmd>{<function>}
Fast for-loop. <list> is not expanded. Entries of <list> will be stored in <cmd> and at every
iteration <function> is executed.
Contributed by Morten Høgholm.
72 \long\def\XKV@for@n#1#2#3{%
73   \XKV@tempa@toks{#1}\edef#2{\the\XKV@tempa@toks}%
74   \ifx#2\@empty
75     \XKV@for@break
76   \else
77     \expandafter\XKV@f@r
78   \fi
79   #2{#3}#1,\@nil,%
80 }

\XKV@f@r  <cmd>{<function>}<entry>,
Looping macro.
81 \long\def\XKV@f@r#1#2#3,{%
82   \XKV@tempa@toks{#3}\edef#1{\the\XKV@tempa@toks}%
83   \ifx#1\@nnil
84     \expandafter\@gobbletwo
85   \else
86     #2\expandafter\XKV@f@r
87   \fi
88   #1{#2}%
89 }

\XKV@for@break  <text>\@nil,
Macro to stop the for-loop.
90 \long\def\XKV@for@break #1\@nil,{\fi}

\XKV@for@o  <listcmd><cmd>{<function>}
<listcmd> is expanded once before starting the loop.
91 \long\def\XKV@for@o#1{\expandafter\XKV@for@n\expandafter{#1}}

\XKV@for@en  {<list>}<cmd>{<function>}
As \XKV@for@n, but this macro will execute <function> also when <list> is empty. This
is done to support packages that use the ‘empty family’, like PSTricks.
92 \long\def\XKV@for@en#1#2#3{\XKV@f@r#2{#3}#1,\@nil,}

\XKV@for@eo  <listcmd><cmd>{<function>}
As \XKV@for@o, but this macro will execute <function> also when <listcmd> is empty.
93 \long\def\XKV@for@eo#1#2#3{%
94   \def#2{\XKV@f@r#2{#3}}\expandafter#2#1,\@nil,%
95 }

```

```

\XKV@whilst <listcmd><cmd><if>\fi{<function>}
<listcmd> is expanded once. Execution of <function> stops when either the list has ran
out of elements or <if> is not true anymore. When using \iftrue for <if>, the execution
of the macro is the same as that of \XKV@for@o, but contains an additional check at
every iteration and is hence less efficient than \XKV@for@o in that situation.
96 \long\def\XKV@whilst#1#2#3\fi#4{%
Check whether the condition is true and start iteration.
97 #3\expandafter\XKV@wh@list#1,\@nil,\@nil\@@#2#3\fi{#4}{}\fi
98 }
\XKV@wh@list <entry>,<text>\@@<cmd><if>\fi{<function>}{<previous>}
Performs iteration and checks extra condition. This macro is not optimized for the case
that the list contains a single element. At the end of every iteration, the current <entry>
will be stored in <previous> for the next iteration. The previous entry is necessary when
stepping out of the loop.
99 \long\def\XKV@wh@list#1,#2\@@#3#4\fi#5#6{%
Define the running <cmd>.
100 \def#3{#1}%
If we find the end of the list, stop.
101 \ifx#3\@nnil
102 \def#3{#6}\expandafter\XKV@wh@l@st
103 \else
If the condition is met, execute <function> and continue. Otherwise, define the running
command to be the previous entry (which inflicted the condition becoming false) and
stop.
104 #4%
105 #5\expandafter\expandafter\expandafter\XKV@wh@list
106 \else
107 \def#3{#6}\expandafter\expandafter\expandafter\XKV@wh@l@st
108 \fi
109 \fi
110 #2\@@#3#4\fi{#5}{#1}%
111 }
\XKV@wh@l@st <text>\@@<cmd><if>\fi{<function>}{<previous>}
Macro to gobble remaining input.
112 \long\def\XKV@wh@l@st#1\@@#2#3\fi#4#5{}
\XKV@addtomacro@n <macro>{<content>}
Adds <content> to <macro> without expanding it.
113 \def\XKV@addtomacro@n#1#2{%
114 \XKV@tempa@toks\expandafter{#1#2}%
115 \edef#1{\the\XKV@tempa@toks}%
116 }
\XKV@addtomacro@o <macro>{<content>}
Adds <content> to <macro> after expanding the first token of <content> once. Often used
to add the content of a macro to another macro.
117 \def\XKV@addtomacro@o#1#2{%
118 \expandafter\XKV@addtomacro@n\expandafter#1\expandafter{#2}%
119 }

```

`\XKV@addtolist@n`  $\langle cmd \rangle \{ \langle content \rangle \}$   
 Adds  $\langle content \rangle$  to the list in  $\langle cmd \rangle$  without expanding  $\langle content \rangle$ . Notice that it is assumed that  $\langle cmd \rangle$  is not undefined.

```
120 \def\XKV@addtolist@n#1#2{%
121   \ifx#1\@empty
122     \XKV@addtomacro@n#1{#2}%
123   \else
124     \XKV@addtomacro@n#1{,#2}%
125   \fi
126 }
```

`\XKV@addtolist@o`  $\langle cmd \rangle \{ \langle content \rangle \}$   
 Adds  $\langle content \rangle$  to the list in  $\langle cmd \rangle$  after expanding the first token in  $\langle content \rangle$  once.

```
127 \def\XKV@addtolist@o#1#2{%
128   \ifx#1\@empty
129     \XKV@addtomacro@o#1#2%
130   \else
131     \XKV@addtomacro@o#1{\expandafter,#2}%
132   \fi
133 }
```

`\XKV@addtolist@x`  $\langle cmd \rangle \{ \langle content \rangle \}$   
 Adds  $\langle content \rangle$  to the list in  $\langle cmd \rangle$  after a full expansion of both  $\langle cmd \rangle$  and  $\langle content \rangle$ .

```
134 \def\XKV@addtolist@x#1#2{\edef#1{#1\ifx#1\@empty\else,\fi#2}}
```

`\@selective@sanitize` [ $\langle level \rangle$ ]  $\{ \langle character string \rangle \} \{ \langle cmd \rangle \}$

`\@selective@sanitize` Converts selected characters, given by  $\langle character string \rangle$ , within the first-level expansion of  $\langle cmd \rangle$  to category code 12, leaving all other tokens (including grouping braces) untouched. Thus, macros inside  $\langle cmd \rangle$  do not lose their function, as it is the case with `\@onelevel@sanitize`. The resulting token list is again saved in  $\langle cmd \rangle$ .

Example: `\def\cs{ ~\fi~}` and `\@selective@sanitize{!~}\cs` will change the catcode of ‘~’ to *other* within `\cs`, while `\fi` and ‘~’ will remain unchanged. As the example shows, unbalanced conditionals are allowed.

Remarks:  $\langle cmd \rangle$  should not contain the control sequence `\bgroup`; however, `\csname bgroup\endcsname` and `\egroup` are possible. The optional  $\langle level \rangle$  command controls up to which nesting level sanitizing takes place inside groups; 0 will only sanitize characters in the top level, 1 will also sanitize within the first level of braces (but not in the second), etc. The default value is 10000.

```
135 \def\@selective@sanitize{\@testopt\@selective@sanitize\M}
136 \def\@selective@sanitize[#1]#2#3{%
137   \begingroup
138     \count#1\relax\advance\count@\@one
139     \XKV@toks\expandafter{#3}%
140     \def#3{#2}\@onelevel@sanitize#3%
141     \edef#3{\the\XKV@toks}%
142     \expandafter\@selective@sanitize\expandafter#3#3%
143     \expandafter\XKV@tempa@toks\expandafter{#3}%
144     \expandafter\endgroup\expandafter\toks@\expandafter{\the\XKV@tempa@toks}%
145     \edef#3{\the\toks@}%
146 }
```



`\@s@l@ctive@sanitize`  $\langle cmd \rangle$   $\langle sanitized\ character\ string \rangle$   $\langle token\ list \rangle$

Performs the main work. Here, the characters in  $\langle sanitized\ character\ string \rangle$  are already converted to catcode 12,  $\langle token\ list \rangle$  is the first-level expansion of the original contents of  $\langle cmd \rangle$ . The macro basically steps through the  $\langle token\ list \rangle$ , inspecting each single token to decide whether it has to be sanitized or passed to the result list. Special care has to be taken to detect spaces, grouping characters and conditionals (the latter may disturb other expressions). However, it is easier and more efficient to look for T<sub>E</sub>X primitives in general – which are characterized by a  $\backslash$  meaning that starts with a backslash – than to test whether a token equals specifically  $\backslash if$ ,  $\backslash else$ ,  $\backslash fi$ , etc. Note that `\@s@l@ctive@sanitize` is being called recursively if  $\langle token\ list \rangle$  contains grouping braces.

```

147 \def \@s@l@ctive@sanitize#1#2#3{%
148   \def \@i{\futurelet \@tok\@ii}%
149   \def \@ii{%
150     \expandafter \@iii\meaning \@tok\relax
151     \ifx \@tok \@s@l@ctive@sanitize
152       \let \@cmd \@gobble
153     \else
154       \ifx \@tok @sptoken
155         \XKV@toks\expandafter{#1}\edef#1{\the\XKV@toks\space}%
156         \def \@cmd{\afterassignment \@i\let \@tok= }%
157       \else
158         \let \@cmd \@iv
159       \fi
160     \fi
161   \@cmd
162 }%
163 \def \@iii##1##2\relax{\if##1\@backslashchar\let \@tok\relax\fi}%
164 \def \@iv##1{%
165   \toks@\expandafter{#1}\XKV@toks{##1}%
166   \ifx \@tok\bgroup
167     \advance\count@ \m@ne
168     \ifnum\count@ > \z@
169       \begingroup
170         \def#1{\expandafter \@s@l@ctive@sanitize
171           \csname\string#1\endcsname{#2}}%
172         \expandafter#1\expandafter{\the\XKV@toks}%
173         \XKV@toks\expandafter\expandafter\expandafter
174           {\csname\string#1\endcsname}%
175         \edef#1{\noexpand\XKV@toks{\the\XKV@toks}}%
176         \expandafter\endgroup#1%
177       \fi
178       \edef#1{\the\toks@{\the\XKV@toks}}%
179       \advance\count@\@ne
180       \let \@cmd \@i
181     \else
182       \edef#1{\expandafter\string\the\XKV@toks}%
183       \expandafter\in@\expandafter{#1}{#2}%
184       \edef#1{\the\toks@\ifin@#1\else
185         \ifx \@tok@sptoken\space\else\the\XKV@toks\fi\fi}%
186       \edef \@cmd{\noexpand \@i\ifx \@tok@sptoken\the\XKV@toks\fi}%
187     \fi
188   \@cmd

```

```

189 }%
190 \let#1\@empty\@i#3\@s@l@ctive@sanitize
191 }

```

`\XKV@checksantizea`  $\{\langle content \rangle\}\langle cmd \rangle$   
 Check whether  $\langle content \rangle$ , to be saved to macro  $\langle cmd \rangle$  unexpanded, contains the characters = or , with wrong catcodes. If so, it sanitizes them before saving  $\langle content \rangle$  to  $\langle cmd \rangle$ .

```

192 \def\XKV@checksantizea#1#2{%
193   \XKV@ch@cksantize{#1}#2=%
194   \ifin@\else\XKV@ch@cksantize{#1}#2,\fi
195   \ifin@\@selective@sanitize[0]{,=#2}\fi
196 }

```

`\XKV@checksantizeb`  $\{\langle content \rangle\}\langle cmd \rangle$   
 Similar to `\XKV@checksantizea`, but only checks commas.

```

197 \def\XKV@checksantizeb#1#2{%
198   \XKV@ch@cksantize{#1}#2,%
199   \ifin@\@selective@sanitize[0],#2\fi
200 }

```

`\XKV@ch@cksantize`  $\{\langle character string \rangle\}\langle cmd \rangle\langle token \rangle$

This macro first checks whether at least one  $\langle token \rangle$  is in  $\langle character string \rangle$ . If that is the case, it checks whether the character has catcode 12. Note that the macro will conclude that the character does not have catcode 12 when it is used inside a group {}, but that is not a problem, as we don't expect  $\langle token \rangle$  (namely , or =) inside a group, unless this group is in a key value. But we won't worry about those characters anyway since the relevant user key macro will have to process that. Further, it is assumed that all occurrences of  $\langle token \rangle$  in  $\langle character string \rangle$  have the same catcode.  $\langle cmd \rangle$  is used as a temporary macro and will contain  $\langle character string \rangle$  at the end of the macro.

```

201 \def\XKV@ch@cksantize#1#2#3{%
202   \XKV@tempa@toks{#1}\edef#2{\the\XKV@tempa@toks}%
203   \@onelevel@sanitize#2%

```

Check whether there is at least one = present.

```

204   \@expandtwoargs\in@#3{#2}%
205   \ifin@

```

If so, try to find it. If we can't find it, the character(s) has (or have) the wrong catcode. In that case sanitizing is necessary. This actually occurs, because the input was read by T<sub>E</sub>X before (and for instance stored in a macro or token register).

```

206   \def#2##1#3##2\@nil{%
207     \XKV@tempa@toks{##2}\edef#2{\the\XKV@tempa@toks}%
208     \ifx#2\@empty\else\in@false\fi
209   }%
210   #2#1#3\@nil
211   \fi
212   \XKV@tempa@toks{#1}\edef#2{\the\XKV@tempa@toks}%
213 }

```

`\XKV@sp@deflist`  $\langle cmd \rangle\{\langle token list \rangle\}$

Defines  $\langle cmd \rangle$  as  $\langle token list \rangle$  after removing spaces surrounding elements of the list in  $\langle token list \rangle$ . So, `key a, key b` becomes `key a, key b`. This is used to remove spaces

from around elements in a list. Using `\zap@space` for this job, would also remove the spaces inside elements and hence changing key or family names with spaces. This method is slower, but does allow for spaces in key and family names, just as `keyval` did. We need this algorithm at several places to be able to perform `\in@{,key,}{, . . . ,}`, without having to worry about spaces in between commas and key names.

```

214 \def\XKV@sp@deflist#1#2{%
215   \let#1\@empty
216   \XKV@for@n{#2}\XKV@resa{%
217     \expandafter\KV@@sp@def\expandafter\XKV@resa\expandafter{\XKV@resa}%
218     \XKV@addtomacro@o#1{\expandafter,\XKV@resa}%
219   }%
220   \ifx#1\@empty\else
221     \def\XKV@resa,##1\@nil{\def#1{##1}}%
222     \expandafter\XKV@resa#1\@nil
223   \fi
224 }

```

`\XKV@merge` *<list>*{*<new items>*}{*<filter>*}

This is a merging macro. For a given new item, the old items are scanned. If an old item key name matches with a new one, the new one will replace the old one. If not, the old one will be appended (and might be overwritten in a following loop). If, at the end of the old item loop the new item has not been used, it will be appended to the end of the list. This macro works irrespective of special syntax. The *<filter>* is used to filter the key name from the syntax, eg `\global{key}`. All occurrences of a particular key in the existing list will be overwritten by the new item. This macro is used to make `\savekeys` and `\presetkeys` incremental. The *<filter>* is `\XKV@getsg` and `\XKV@getkeyname` respectively.

```

225 \def\XKV@merge#1#2#3{%
226   \XKV@checksantizea{#2}\XKV@tempa

```

Start the loop over the new presets. At every iteration, one new preset will be compared with old presets.

```

227   \XKV@for@o\XKV@tempa\XKV@tempa{%
228     \XKV@pltrue

```

Retrieve the key name of the new item at hand.

```

229     #3\XKV@tempa\XKV@tempb

```

Store the (partially updated) old list in a temp macro and empty the original macro.

```

230     \let\XKV@tempc#1%
231     \let#1\@empty

```

Start a loop over the old list.

```

232     \XKV@for@o\XKV@tempc\XKV@tempc{%

```

Retrieve the key name of the old key at hand.

```

233         #3\XKV@tempc\XKV@tempd
234         \ifx\XKV@tempb\XKV@tempd

```

If the key names are equal, append the new item to the list and record that this key should not be added to the end of the presets list.

```

235             \XKV@plfalse
236             \XKV@addtolist@o#1\XKV@tempa
237         \else

```

If the key names are not equal, then just append the current item to the list.

```
238     \XKV@addtolist@o#1\XKV@tempc
239     \fi
240   }%
```

If, after checking the old item, no old item has been overwritten then append the new item to the end of the existing list.

```
241   \ifXKV@pl\XKV@addtolist@o#1\XKV@tempa\fi
242 }%
```

If requested, save the new list globally.

```
243 \ifXKV@st\global\let#1#1\fi
244 }
```

`\XKV@delete` *<list>*{*<delete>*}{*<filter>*}

Delete entries *<delete>* by key name from a *<list>* of presets or save keys using *<filter>*. For `\delpresetkeys`, this is the macro `\XKV@getkeyname` and for `\delsavekeys`, it is the macro `\XKV@getsg`.

```
245 \def\XKV@delete#1#2#3{%
```

Sanitize comma's.

```
246 \XKV@checksanitizetb{#2}\XKV@tempa
```

Copy the current list and make the original empty.

```
247 \let\XKV@tempb#1%
```

```
248 \let#1\@empty
```

Run over the current list.

```
249 \XKV@for@o\XKV@tempb\XKV@tempb{%
```

Get the key name to identify the current entry.

```
250   #3\XKV@tempb\XKV@tempc
```

If the current key name is in the list, do not add it anymore.

```
251   \@expandtwoargs\in@{,\XKV@tempc,}-{,\XKV@tempa,}%
```

```
252   \ifin@else\XKV@addtolist@o#1\XKV@tempb\fi
```

```
253 }%
```

Save globally is necessary.

```
254 \ifXKV@st\global\let#1#1\fi
```

```
255 }
```

`\XKV@warn` Warning and error macros. We redefine the keyval error macros to use the xkeyval ones.

`\XKV@err` This avoids redefining them again when we redefine the `\XKV@warn` and `\XKV@err` macros in `xkeyval.sty`.

```
\KV@errx 256 \def\XKV@warn#1{\message{xkeyval warning: #1}}
```

```
257 \def\XKV@err#1{\errmessage{xkeyval error: #1}}
```

```
258 \def\KV@errx{\XKV@err}
```

```
259 \let\KV@err\KV@errx
```

`\XKV@ifstar` Checks whether the following token is a \* or +. Use `\XKV@ifnextchar` to perform the action safely and ignore catcodes.

```
\XKV@ifplus 260 \def\XKV@ifstar#1{\@ifnextcharacter*{\@firstoftwo{#1}}}
```

```
261 \def\XKV@ifplus#1{\@ifnextcharacter+{\@firstoftwo{#1}}}
```

```

\XKV@makepf  {<prefix>}
This macro creates the prefix, like prefix@ in \prefix@family@key. First it deletes
spaces from the input and checks whether it is empty. If not empty, an @-sign is added.
The use of the XKV prefix is forbidden to protect internal macros and special macros
like saved key values.
262 \def\XKV@makepf#1{%
263   \KV@@sp@def\XKV@prefix{#1}%
264   \def\XKV@resa{XKV}%
265   \ifx\XKV@prefix\XKV@resa
266     \XKV@err{'XKV' prefix is not allowed}%
267     \let\XKV@prefix\@empty
268   \else
269     \edef\XKV@prefix{\ifx\XKV@prefix\@empty\else\XKV@prefix @\fi}%
270   \fi
271 }

\XKV@makehd  {<family>}
Creates the header, like prefix@family@ in \prefix@family@key. If <family> is
empty, the header reduces to prefix@.
272 \def\XKV@makehd#1{%
273   \expandafter\KV@@sp@def\expandafter\XKV@header\expandafter{#1}%
274   \edef\XKV@header{%
275     \XKV@prefix\ifx\XKV@header\@empty\else\XKV@header @\fi
276   }%
277 }

\XKV@srstate  {<postfix1>}{<postfix2>}
Macro to save and restore xkeyval internals to allow for nesting \setkeys com-
mands. It executes a for loop over a set of xkeyval internals and does, for instance,
\let\XKV@na@i\XKV@na to prepare for stepping a level deeper. If <prefix2> is empty,
we step a level deeper. If <prefix1> is empty, we go a level up. The non-empty argument
is always @\romannumeral\XKV@depth. Notice that this also helps to keep changes to
boolean settings (for instance by \XKV@cc*) local to the execution of that key.
278 \def\XKV@srstate#1#2{%
279   \ifx\@empty#2\@empty\advance\XKV@depth\@ne\fi
280   \XKV@for@n{\XKV@prefix,XKV@fams,XKV@tkey,XKV@na,%
281     \ifXKV@st,\ifXKV@pl,\ifXKV@knf}\XKV@resa{%
282     \expandafter\let\csname\XKV@resa#1\expandafter
283       \endcsname\csname\XKV@resa#2\endcsname
284   }%
285   \ifx\@empty#1\@empty\advance\XKV@depth\m@ne\fi
286 }

\XKV@testopta  {<function>}
\XKV@t@stopta Tests for the presence of an optional star or plus and executes <function> afterwards.
287 \def\XKV@testopta#1{%
288   \XKV@ifstar{\XKV@sttrue\XKV@t@stopta{#1}}%
289   {\XKV@stfalse\XKV@t@stopta{#1}}%
290 }
291 \def\XKV@t@stopta#1{\XKV@ifplus{\XKV@pltrue#1}{\XKV@plfalse#1}}

```

```

\XKV@testoptb  {\function}
\XKV@t@stoptb First check for an optional prefix. Afterwards, set the <prefix>, set the header, remove
spaces from the <family> and execute <function>.
292 \def\XKV@testoptb#1{\@testopt{\XKV@t@stoptb#1}{KV}}
293 \def\XKV@t@stoptb#1[#2]#3{%
Set prefix.
294 \XKV@makepf{#2}%
Set header.
295 \XKV@makehd{#3}%
Save family name for later use.
296 \KV@sp@def\XKV@tfam{#3}%
297 #1%
298 }

\XKV@testoptc  {\function}
\XKV@t@stoptc Test for an optional <prefix>. Then, set the <prefix>, sanitize comma's in the list of
<families> and remove redundant spaces from this list. Finally, check for optional key
names that should not be set and execute <function>.
299 \def\XKV@testoptc#1{\@testopt{\XKV@t@stoptc#1}{KV}}
300 \def\XKV@t@stoptc#1[#2]#3{%
301 \XKV@makepf{#2}%
302 \XKV@checksanitizib{#3}\XKV@fams
303 \expandafter\XKV@sp@deflist\expandafter
304 \XKV@fams\expandafter{\XKV@fams}%
305 \@testopt#1{}}%
306 }

\XKV@testoptd  {\function}
\XKV@t@stoptd Use \XKV@testoptb first to find <prefix> and the <family>. Then check for optional
<mp> ('macro prefix'). Next eat the <key> name and check for an optional <default>
value.
307 \def\XKV@testoptd#1#2{%
308 \XKV@testoptb{%
309 \edef\XKV@tempa{#2\XKV@header}%
310 \def\XKV@tempb{\@testopt{\XKV@t@stoptd#1}}%
311 \expandafter\XKV@tempb\expandafter{\XKV@tempa}%
312 }%
313 }
314 \def\XKV@t@stoptd#1[#2]#3{%
\ifXKV@st gives the presence of an optional default value.
315 \@ifnextchar[{\XKV@sttrue#1{#2}{#3}}{\XKV@stfalse#1{#2}{#3}[]}%
316 }

\XKV@ifcmd  {\tokens}{\macro}{\cmd}{\yes}{\no}
\XKV@ifcmd This macro checks whether the <tokens> contains the macro specification <macro>. If
so, the argument to this macro will be saved to <cmd> and <yes> will be executed. Other-
wise, the content of <tokens> is saved to <cmd> and <no> is executed. This macro will, for
instance, be used to distinguish key and \global{key} and retrieve key in the latter
case.
317 \def\XKV@ifcmd#1#2#3{%

```

```

318 \def\XKV@ifcmd##1#2##3\@nil##4{%
319 \def##4{##2}\ifx##4\@nil
320 \def##4{##1}\expandafter\@secondoftwo
321 \else
322 \expandafter\@firstoftwo
323 \fi
324 }%
325 \XKV@ifcmd#1#2{\@nil}\@nil#3%
326 }

```

`\XKV@getkeyname`  $\langle keyvalue \rangle \langle bin \rangle$

Utility macro to retrieve the key name from  $\langle keyvalue \rangle$  which is of the form `key=value`, `\savevalue{key}=value` or `\gsavevalue{key}=value`, possibly without value. `\ifXKV@rkv` will record whether this particular value should be saved. `\ifXKV@sg` will record whether this value should be saved globally or not. The key name will be stored in  $\langle bin \rangle$ .

```

327 \def\XKV@getkeyname#1#2{\expandafter\XKV@g@tkeyname#1=\@nil#2}

```

`\XKV@g@tkeyname`  $\langle key \rangle = \langle value \rangle \backslash @nil \langle bin \rangle$

Use `\XKV@ifcmd` several times to check the syntax of  $\langle value \rangle$ . Save  $\langle key \rangle$  to  $\langle bin \rangle$ .

```

328 \def\XKV@g@tkeyname#1=#2\@nil#3{%
329 \XKV@ifcmd{#1}\savevalue#3{\XKV@rkvtrue\XKV@sgfalse}{%
330 \XKV@ifcmd{#1}\gsavevalue#3%
331 {\XKV@rkvtrue\XKV@sgtrue}{\XKV@rkvfalse\XKV@sgfalse}}%
332 }%
333 }

```

`\XKV@getsg`  $\langle key \rangle \langle bin \rangle$

Utility macro to check whether `key` or `\global{key}` has been specified in  $\langle key \rangle$ . The key name is saved to  $\langle bin \rangle$

```

334 \def\XKV@getsg#1#2{%
335 \expandafter\XKV@ifcmd\expandafter{#1}\global#2\XKV@sgtrue\XKV@sgfalse
336 }

```

`\XKV@define@default`  $\{ \langle key \rangle \} \{ \langle default \rangle \}$

Defines the default value macro for  $\langle key \rangle$  and given `\XKV@header`.

```

337 \def\XKV@define@default#1#2{%
338 \expandafter\def\csname\XKV@header#1@default\expandafter
339 \endcsname\expandafter{\csname\XKV@header#1\endcsname{#2}}}%
340 }

```

`\define@key`  $[ \langle prefix \rangle ] \{ \langle family \rangle \}$

Macro to define a key in a family. Notice the use of the KV prefix as default prefix. This is done to allow setting both `keyval` and `xkeyval` keys with a single command. This top level command first checks for an optional  $\langle prefix \rangle$  and the mandatory  $\langle family \rangle$ .

```

341 \def\define@key{\XKV@testoptb\XKV@define@key}

```

`\XKV@define@key`  $\{ \langle key \rangle \}$

Check for an optional default value. If none present, define the key macro, else continue to eat the default value.

```

342 \def\XKV@define@key#1{%
343 \@ifnextchar[{\XKV@d@f@i@n@e@k@y{#1}}]{%

```

```

344     \expandafter\def\csname\XKV@header#1\endcsname###1%
345   }%
346 }

\XKV@define@key {<key>} [ <default> ]
Defines the key macro and the default value macro.
347 \def\XKV@define@key#1[#2]{%
348   \XKV@define@default{#1}{#2}%
349   \expandafter\def\csname\XKV@header#1\endcsname###1%
350 }

\define@cmdkey [ <prefix> ] { <family> } [ <mp> ] { <key> }
Define a command key. Test for optional <prefix>, mandatory <family>, optional <mp>
'macro prefix' and mandatory <key> name.
351 \def\define@cmdkey{\XKV@testoptd\XKV@define@cmdkey{cmd}}

\XKV@define@cmdkey { <mp> } { <key> } [ <default> ] { <function> }
Define the default value macro and the key macro. The key macro first defines the
control sequence formed by the <mp> and <key> to expand to the user input and then
executes the <function>.
352 \def\XKV@define@cmdkey#1#2[#3]#4{%
353   \ifXKV@st\XKV@define@default{#2}{#3}\fi
354   \def\XKV@tempa{\expandafter\def\csname\XKV@header#2\endcsname###1}%
355   \begingroup\expandafter\endgroup\expandafter\XKV@tempa\expandafter
356     {\expandafter\def\csname#1#2\endcsname{##1}#4}%
357 }

\define@cmdkeys [ <prefix> ] { <family> } [ <mp> ] { <keys> }
Define multiple command keys.
358 \def\define@cmdkeys{\XKV@testoptd\XKV@define@cmdkeys{cmd}}

\XKV@define@cmdkeys { <mp> } { <keys> } [ <default> ]
Loop over <keys> and define a command key for every entry.
359 \def\XKV@define@cmdkeys#1#2[#3]{%
360   \XKV@sp@deflist\XKV@tempa{#2}%
361   \XKV@for@o\XKV@tempa\XKV@tempa{%
362     \edef\XKV@tempa{\noexpand\XKV@define@cmdkey{#1}{\XKV@tempa}}%
363     \XKV@tempa[#3]{}%
364   }%
365 }

\define@choicekey *+ [ <prefix> ] { <family> }
Choice keys. First check optional star, plus and prefix and store the family.
366 \def\define@choicekey{\XKV@testopta{\XKV@testoptb\XKV@define@choicekey}}

\XKV@define@choicekey { <key> }
Check for optional storage bins for the input and the number of the input in the list of
allowed inputs.
367 \def\XKV@define@choicekey#1{\@testopt{\XKV@define@choicekey{#1}}{}}

```



```

\XKV@d@fine@choicekey  {<key>} [<bin>] {<allowed>}
Store the storage bin and the list of allowed inputs for later use. After that, check for an
optional default value.
368 \def\XKV@d@fine@choicekey#1[#2]#3{%
369   \toks@{#2}%
370   \XKV@sp@deflist\XKV@tempa{#3}\XKV@toks\expandafter{\XKV@tempa}%
371   \@ifnextchar[{\XKV@d@fine@ch@icekey{#1}}{\XKV@d@fine@ch@ic@key{#1}}%
372 }

\XKV@d@fine@ch@icekey  {<key>} [<default>]
Define the default value macro if a default value was specified.
373 \def\XKV@d@fine@ch@icekey#1[#2]{%
374   \XKV@define@default{#1}{#2}%
375   \XKV@d@fine@ch@ic@key{#1}%
376 }

\XKV@d@fine@ch@ic@key  {<key>}
Eat correct number of arguments.
377 \def\XKV@d@fine@ch@ic@key#1{%
378   \ifXKV@pl\XKV@afterelsefi
379     \expandafter\XKV@d@f@ne@ch@ic@k@y
380   \else\XKV@afterfi
381     \expandafter\XKV@d@f@ne@ch@ic@key
382   \fi
383   \csname\XKV@header#1\endcsname
384 }

\XKV@d@f@ne@ch@ic@key  <key macro>{<function>}
Eat one argument and pass it on to the macro that will define the key macro.
385 \def\XKV@d@f@ne@ch@ic@key#1#2{\XKV@d@f@n@ch@ic@k@y#1{#2}}

\XKV@d@f@ne@ch@ic@k@y  <key macro>{<function1>}{<function2>}
Eat two arguments and pass these on to the macro that will define the key macro.
<function1> will be executed on correct input, <function2> on incorrect input.
386 \def\XKV@d@f@ne@ch@ic@k@y#1#2#3{\XKV@d@f@n@ch@ic@k@y#1{#2}{#3}}

\XKV@d@f@n@ch@ic@k@y  <key macro>{<function>}
Create the key macros. \XKV@checkchoice will be used to check the choice and exe-
cute one of its mandatory arguments.
387 \def\XKV@d@f@n@ch@ic@k@y#1#2{%
388   \edef#1##1{%
389     \ifXKV@st\noexpand\XKV@sttrue\else\noexpand\XKV@stfalse\fi
390     \ifXKV@pl\noexpand\XKV@pltrue\else\noexpand\XKV@plfalse\fi
391     \noexpand\XKV@checkchoice[\the\toks@]{#1}{\the\XKV@toks}%
392   }%
393   \def\XKV@tempa{\def#1###1}%
394   \expandafter\XKV@tempa\expandafter{#1{#1}#2}%
395 }

\define@boolkey  + [<prefix>] {<family>} [<mp>] {<key>}
Define a boolean key. This macro checks for an optional +, an optional <prefix>, the
mandatory <family>, an optional <mp> ('macro prefix') and the mandatory <key> name.
396 \def\define@boolkey{\XKV@t@stopta{\XKV@t@stoptd\XKV@define@boolkey{}}

```

`\XKV@define@boolkey` `{\mp}{\key}[\default]`  
Decide to eat 1 or 2 mandatory arguments for the key macro. Further, construct the control sequence for the key macro and the one for the if.

```

397 \def\XKV@define@boolkey#1#2[#3]{%
398   \ifXKV@pl\XKV@afterelsefi
399   \expandafter\XKV@d@f@ne@boolkey
400   \else\XKV@afterfi
401   \expandafter\XKV@d@f@ine@boolkey
402   \fi
403   \csname\XKV@header#2\endcsname{#2}{#1#2}{#3}%
404 }
```

`\XKV@d@f@ine@boolkey` `\key macro}{\key}{\ifname}{\default}{\function}`  
Eat one mandatory key function and pass it. Insert 'setting the if'.

```

405 \def\XKV@d@f@ine@boolkey#1#2#3#4#5{%
406   \XKV@d@f@ne@b@olkey#1{#2}{#3}{#4}%
407   {\csname#3\XKV@resa\endcsname#5}%
408 }
```

`\XKV@d@f@ne@boolkey` `\key macro}{\key}{\ifname}{\default}{\func1}{\func2}`  
Eat two mandatory key functions and pass them. Insert 'setting the if'.

```

409 \def\XKV@d@f@ne@boolkey#1#2#3#4#5#6{%
410   \XKV@d@f@ne@b@olkey#1{#2}{#3}{#4}%
411   {\csname#3\XKV@resa\endcsname#5}{#6}%
412 }
```

`\XKV@d@f@ne@b@olkey` `\key macro}{\key}{\ifname}{\default}{\function}`  
Create the if, the default value macro (if a default value was present) and the key macro. We use `\XKV@checkchoice` internally to check the input and `\XKV@resa` to store the user input and pass it to setting the conditional.

```

413 \def\XKV@d@f@ne@b@olkey#1#2#3#4#5{%
414   \expandafter\newif\csname if#3\endcsname
415   \ifXKV@st\XKV@define@default{#2}{#4}\fi
416   \ifXKV@pl
417     \def##1{\XKV@pltrue\XKV@sttrue
418       \XKV@checkchoice[\XKV@resa]{##1}{true,false}#5%
419     }%
420   \else
421     \def##1{\XKV@plfalse\XKV@sttrue
422       \XKV@checkchoice[\XKV@resa]{##1}{true,false}#5%
423     }%
424   \fi
425 }
```

`\define@boolkeys` `[\prefix]{\family}[\mp]{\keys}`  
Define multiple boolean keys without user specified key function. The key will, of course, still set the if with user input.

```

426 \def\define@boolkeys{\XKV@plfalse\XKV@testoptd\XKV@define@boolkeys{}}
```

`\XKV@define@boolkeys` `{\mp}{\keys}[\default]`  
Loop over the list of `\keys` and create a boolean key for every entry.

```

427 \def\XKV@define@boolkeys#1#2[#3]{%
```

```

428 \XKV@sp@deflist\XKV@tempa{#2}%
429 \XKV@for@o\XKV@tempa\XKV@tempa{%
430 \expandafter\XKV@d@fine@boolkeys\expandafter{\XKV@tempa}{#1}{#3}%
431 }%
432 }

```

`\XKV@d@fine@boolkeys` `{<key>}{<mp>}{<default>}`

Use `\XKV@d@f@one@b@olkey` internally to define the if, the default value macro (if present) and the key macro.

```

433 \def\XKV@d@fine@boolkeys#1#2#3{%
434 \expandafter\XKV@d@f@one@b@olkey\csname\XKV@header#1\endcsname
435 {#1}{#2#1}{#3}{\csname#2#1\XKV@resa\endcsname}}%
436 }

```

`\XKV@cc` This macro is used inside key macros to perform input checks. This is the user interface to `\XKV@checkchoice` and we only use the latter internally to avoid slow parsings of optional \* and +.

```

437 \def\XKV@cc{\XKV@testopta{\@testopt\XKV@checkchoice{}}}

```

`\XKV@checkchoice` `[<bin>]{<input>}{<allowed>}`

Checks whether `<bin>` contains at least one control sequence and converts `<input>` and `<allowed>` to lowercase if requested. If `<bin>` is empty, perform the fast `\in@` check immediately. Else, determine whether the bin contains one or two tokens. For the first alternative, we can still use the fast `\in@` check. Notice that this macro uses settings for `\ifXKV@st` and `\ifXKV@pl`.

```

438 \def\XKV@checkchoice[#1]#2#3{%
439 \def\XKV@tempa{#1}%
440 \ifXKV@st\lowercase{\fi
441 \ifx\XKV@tempa\@empty
442 \def\XKV@tempa{\XKV@ch@ckch@ice\@nil{#2}{#3}}%
443 \else
444 \def\XKV@tempa{\XKV@ch@ckchoice#1\@nil{#2}{#3}}%
445 \fi
446 \ifXKV@st}\fi\XKV@tempa
447 }

```

`\XKV@ch@ckchoice` `<bin1><bin2>\@nil{<input>}{<allowed>}`

Check whether `<bin2>` is empty. In that case, only the `<input>` should be saved and we can continue with the fast `\in@` check. If not, also the number of the input in the `<allowed>` list should be saved and we need to do a slower while type of loop.

```

448 \def\XKV@ch@ckchoice#1#2\@nil#3#4{%
449 \def\XKV@tempa{#2}%
450 \ifx\XKV@tempa\@empty\XKV@afterelsefi
451 \XKV@ch@ckch@ice#1{#3}{#4}%
452 \else\XKV@afterfi
453 \XKV@ch@ckchoice#1#2{#3}{#4}%
454 \fi
455 }

```

`\XKV@ch@ckch@ice` `<bin>{<input>}{<allowed>}`

Checks whether `<input>` is in the list `<allowed>` and perform actions accordingly.

```

456 \def\XKV@ch@ckch@ice#1#2#3{%
457 \def\XKV@tempa{#1}%

```

If we have a *<bin>*, store the input there.

```
458 \ifx\XKV@tempa\@nnil\let\XKV@tempa\@empty\else
459 \def\XKV@tempa{\def#1{#2}}%
460 \fi
461 \in@{, #2, }{, #3, }%
462 \ifin@
```

The *<input>* is allowed.

```
463 \ifXKV@pl
```

If we have a +, there are two functions. Execute the first.

```
464 \XKV@addtomacro@n\XKV@tempa\@firstoftwo
465 \else
```

Else, we have one function; execute it.

```
466 \XKV@addtomacro@n\XKV@tempa\@firstofone
467 \fi
468 \else
```

If we have a +, there are two functions. Execute the second.

```
469 \ifXKV@pl
470 \XKV@addtomacro@n\XKV@tempa\@secondoftwo
471 \else
```

Else, raise an error and gobble the one function.

```
472 \XKV@toks{#2}%
473 \XKV@err{value ‘\the\XKV@toks’ is not allowed}%
474 \XKV@addtomacro@n\XKV@tempa\@gobble
475 \fi
476 \fi
477 \XKV@tempa
478 }
```

```
\XKV@@ch@ckchoice <bin1><bin2>{<input>}{<allowed>}
```

Walk over the *<allowed>* list and compare each entry with the *<input>*. The input is saved in *<bin1>*, the number of the *<input>* in the *<allowed>* list (starting at zero) is saved in *<bin2>*. If the *<input>* is not allowed, *<bin2>* will be defined to contain -1.

```
479 \def\XKV@@ch@ckchoice#1#2#3#4{%
```

Save the current value of the counter as to avoid disturbing it. We don't use a group as that takes a lot of memory and requires some more tokens (for global definitions).

```
480 \edef\XKV@tempa{\the\count@}\count@\z@
```

The input.

```
481 \def\XKV@tempb{#3}%
```

Define the while loop.

```
482 \def\XKV@tempc##1,{%
483 \def#1{##1}%
484 \ifx#1\@nnil
```

The *<input>* was not in *<allowed>*. Set the number to -1.

```
485 \def#1{#3}\def#2{-1}\count@\XKV@tempa
486 \ifXKV@pl
```

Execute the macro for the case that input was not allowed.

```
487 \let\XKV@tempd\@secondoftwo
488 \else
```

If that function does not exist, raise a generic error and gobble the function to be executed on good input.

```

489     \XKV@toks{#3}%
490     \XKV@err{value ‘\the\XKV@toks’ is not allowed}%
491     \let\XKV@tempd@gobble
492     \fi
493   \else
494     \ifx#1\XKV@tempb

```

We found *<input>* in *<allowed>*. Save the number of the *<input>* in the list *<allowed>*.

```

495     \edef#2{\the\count@}\count@\XKV@tempa
496     \ifXKV@pl
497       \let\XKV@tempd\XKV@@ch@ckch@ice
498     \else
499       \let\XKV@tempd\XKV@@ch@ckch@ic@
500     \fi
501   \else

```

Increase counter and check next item in the list *<allowed>*.

```

502     \advance\count@\@ne
503     \let\XKV@tempd\XKV@tempc
504   \fi
505 \fi
506 \XKV@tempd
507 }%

```

Start the while loop.

```

508 \XKV@tempc#4,\@nil,%
509 }

```

`\XKV@@ch@ckch@ice` *<text>*\@nil,  
`\XKV@@ch@ckch@ic@` Gobble remaining *<text>* and execute the proper key function.

```

510 \def\XKV@@ch@ckch@ice#1\@nil,{\@firstoftwo}
511 \def\XKV@@ch@ckch@ic@#1\@nil,{\@firstofone}

```

`\key@ifundefined` This macro allows checking if a key is defined in a family from a list of families. Check for an optional prefix.

```

512 \def\key@ifundefined{\@testopt\XKV@key@ifundefined{KV}}

```

`\XKV@key@ifundefined` [*<prefix>*]{*<fams>*}

This macro is split in two parts so that `\XKV@p@x` can use only the main part of the macro. First we save the prefix and the list of families.

```

513 \def\XKV@key@ifundefined[#1]#2{%
514   \XKV@makepf{#1}%
515   \XKV@checksanitizeb{#2}\XKV@fams
516   \expandafter\XKV@sp@deflist\expandafter
517   \XKV@fams\expandafter{\XKV@fams}%
518   \XKV@key@if@ndefined
519 }

```

`\XKV@key@if@ndefined` {*<key>*}

Loop over the list of families until we find the key in a family.

```

520 \def\XKV@key@if@ndefined#1{%
521   \XKV@knftrue
522   \KV@@sp@def\XKV@tkey{#1}%

```

Loop over possible families.

```
523 \XKV@whilst\XKV@fams\XKV@tfam\ifXKV@knf\fi{%
```

Set the header.

```
524 \XKV@makehd\XKV@tfam
```

Check whether the macro for the key is defined.

```
525 \XKV@ifundefined{\XKV@header\XKV@tkey}{-}{\XKV@knffalse}%  
526 }%
```

Execute one of the final two arguments depending on state of \XKV@knf.

```
527 \ifXKV@knf  
528 \expandafter\@firstoftwo  
529 \else  
530 \expandafter\@secondoftwo  
531 \fi  
532 }
```

```
\disable@keys [<prefix>]{<family>}
```

Macro that make a key produce a warning on use.

```
533 \def\disable@keys{\XKV@testoptb\XKV@disable@keys}
```

```
\XKV@disable@keys {<keys>}
```

Workhorse for \disable@keys which redefines a list of key macro to produce a warning.

```
534 \def\XKV@disable@keys#1{%  
535 \XKV@checksanitizeb{#1}\XKV@tempa  
536 \XKV@for@o\XKV@tempa\XKV@tempa{%  
537 \XKV@ifundefined{\XKV@header\XKV@tempa}{-}{%  
538 \XKV@err{key ‘\XKV@tempa’ undefined}%  
539 }-%  
540 \edef\XKV@tempb{%  
541 \noexpand\XKV@warn{key ‘\XKV@tempa’ has been disabled}%  
542 }-%  
543 \XKV@ifundefined{\XKV@header\XKV@tempa @default}{-}{%  
544 \edef\XKV@tempc{\noexpand\XKV@define@key{\XKV@tempa}}-%  
545 }-%  
546 \edef\XKV@tempc{\noexpand\XKV@define@key{\XKV@tempa} [] }-%  
547 }-%  
548 \expandafter\XKV@tempc\expandafter{\XKV@tempb}%  
549 }-%  
550 }-%  
551 }
```

```
\presetkeys [<prefix>]{<family>}
```

```
\gpresetkeys
```

This provides the presetting system. The macro works incrementally: keys that have been preset before will overwrite the old preset values, new ones will be added to the end of the preset list.

```
552 \def\presetkeys{\XKV@stfalse\XKV@testoptb\XKV@presetkeys}  
553 \def\gpresetkeys{\XKV@sttrue\XKV@testoptb\XKV@presetkeys}
```

```
\XKV@presetkeys {<head presets>}{<tail presets>}
```

Execute the merging macro \XKV@pr@setkeys for both head and tail presets.

```
554 \def\XKV@presetkeys#1#2{%
```

```

555 \XKV@pr@setkeys{#1}{preseth}%
556 \XKV@pr@setkeys{#2}{presett}%
557 }

```

`\XKV@pr@setkeys` {*<presets>*}{*<postfix>*}

Check whether presets have already been defined. If not, define them and do not start the merging macro. Otherwise, create the control sequence that stores these presets and start merging.

```

558 \def\XKV@pr@setkeys#1#2{%
559 \XKV@ifundefined{XKV@XKV@header#2}{%
560 \XKV@checksantizea{#1}\XKV@tempa
561 \ifXKV@st\expandafter\global\fi\expandafter\def\csname
562 XKV@XKV@header#2\expandafter\endcsname\expandafter{\XKV@tempa}%
563 }{%
564 \expandafter\XKV@merge\csname XKV@XKV@header
565 #2\endcsname{#1}\XKV@getkeyname
566 }%
567 }

```

`\delpresetkeys` [*<prefix>*]{*<family>*}

`\gdelpresetkeys` Macros to remove entries from presets.

```

568 \def\delpresetkeys{\XKV@stfalse\XKV@testoptb\XKV@delpresetkeys}
569 \def\gdelpresetkeys{\XKV@sttrue\XKV@testoptb\XKV@delpresetkeys}

```

`\XKV@delpresetkeys` {*<head key list>*}{*<tail key list>*}

Run the main macro for both head and tail presets.

```

570 \def\XKV@delpresetkeys#1#2{%
571 \XKV@d@lpresetkeys{#1}{preseth}%
572 \XKV@d@lpresetkeys{#2}{presett}%
573 }

```

`\XKV@d@lpresetkeys` {*<key list>*}{*<postfix>*}

Check whether presets have been saved and if so, start deletion algorithm. Supply the macro `\XKV@getkeyname` to retrieve key names from entries.

```

574 \def\XKV@d@lpresetkeys#1#2{%
575 \XKV@ifundefined{XKV@XKV@header#2}{%
576 \XKV@err{no presets defined for ‘\XKV@header’}%
577 }{%
578 \expandafter\XKV@delete\csname XKV@XKV@header
579 #2\endcsname{#1}\XKV@getkeyname
580 }%
581 }

```

`\unpresetkeys` [*<prefix>*]{*<family>*}

`\gunpresetkeys` Removes presets for a particular family.

```

582 \def\unpresetkeys{\XKV@stfalse\XKV@testoptb\XKV@unpresetkeys}
583 \def\gunpresetkeys{\XKV@sttrue\XKV@testoptb\XKV@unpresetkeys}

```

`\XKV@unpresetkeys`

Undefine the preset macros. We make them undefined since this will make them appear undefined to both versions of the macro `\XKV@ifundefined`. Making the macros `\relax` would work in the case that no  $\varepsilon$ -TeX is available (hence using `\ifx\csname`), but doesn't work when  $\varepsilon$ -TeX is used (and using `\ifcsname`).

```

584 \def\XKV@unpresetkeys{%
585   \XKV@ifundefined{XKV@\XKV@header preseth}{%
586     \XKV@err{no presets defined for ‘\XKV@header’}%
587   }{%
588     \ifXKV@st\expandafter\global\fi\expandafter\let
589       \csname XKV@\XKV@header preseth\endcsname\@undefined
590     \ifXKV@st\expandafter\global\fi\expandafter\let
591       \csname XKV@\XKV@header presett\endcsname\@undefined
592   }%
593 }

```

`\savekeys` [*⟨prefix⟩*]{*⟨family⟩*}

`\gsavekeys` Store a list of keys of a family that should always be saved. The macro works incrementally and avoids duplicate entries in the list.

```

594 \def\savekeys{\XKV@stfalse\XKV@testoptb\XKV@savekeys}
595 \def\gsavekeys{\XKV@sttrue\XKV@testoptb\XKV@savekeys}

```

`\XKV@savekeys` {*⟨key list⟩*}

Check whether something has been saved before. If not, start merging.

```

596 \def\XKV@savekeys#1{%
597   \XKV@ifundefined{XKV@\XKV@header save}{%
598     \XKV@checksanitizeb{#1}\XKV@tempa
599     \ifXKV@st\expandafter\global\fi\expandafter\def\csname XKV@%
600       \XKV@header save\expandafter\endcsname\expandafter{\XKV@tempa}%
601   }{%
602     \expandafter\XKV@merge\csname XKV@\XKV@header
603       save\endcsname{#1}\XKV@getsg
604   }%
605 }

```

`\delsavekeys` [*⟨prefix⟩*]{*⟨family⟩*}

`\gdelsavekeys` Remove entries from the list of save keys.

```

606 \def\delsavekeys{\XKV@stfalse\XKV@testoptb\XKV@delsavekeys}
607 \def\gdelsavekeys{\XKV@sttrue\XKV@testoptb\XKV@delsavekeys}

```

`\XKV@delsavekeys` {*⟨key list⟩*}

Check whether save keys are defined and if yes, start deletion algorithm. Use the macro `\XKV@getsg` to retrieve key names from entries.

```

608 \def\XKV@delsavekeys#1{%
609   \XKV@ifundefined{XKV@\XKV@header save}{%
610     \XKV@err{no save keys defined for ‘\XKV@header’}%
611   }{%
612     \expandafter\XKV@delete\csname XKV@\XKV@header
613       save\endcsname{#1}\XKV@getsg
614   }%
615 }

```

`\unsavekeys` [*⟨prefix⟩*]{*⟨family⟩*}

`\gunsavekeys` Similar to `\unpresetkeys`, but removes the ‘save keys list’ for a particular family.

```

616 \def\unsavekeys{\XKV@stfalse\XKV@testoptb\XKV@unsavekeys}
617 \def\gunsavekeys{\XKV@sttrue\XKV@testoptb\XKV@unsavekeys}

```



`\XKV@unsavekeys` Workhorse for `\unsavekeys`.

```

618 \def\XKV@unsavekeys{%
619   \XKV@ifundefined{XKV@\XKV@header save}{%
620     \XKV@err{no save keys defined for ‘\XKV@header’}%
621   }{%
622     \ifXKV@st\expandafter\global\fi\expandafter\let
623       \csname XKV@\XKV@header save\endcsname\undefined
624   }%
625 }
```

`\setkeys` `*+ [⟨prefix⟩] {⟨families⟩}`  
 Set keys. The starred version does not produce errors, but appends keys that cannot be located to the list in `\XKV@rm`. The plus version sets keys in all families that are supplied.

```
626 \def\setkeys{\XKV@testopta{\XKV@testoptc\XKV@setkeys}}
```

`\XKV@setkeys` `[⟨na⟩] {⟨key=value list⟩}`

Workhorse for `\setkeys`.

```

627 \def\XKV@setkeys[#1]#2{%
628   \XKV@checksantizea{#2}\XKV@resb
629   \let\XKV@naa\@empty
```

Retrieve a list of key names from the user input.

```

630   \XKV@for@o\XKV@resb\XKV@tempa{%
631     \expandafter\XKV@g@tkeyname\XKV@tempa=\@nil\XKV@tempa
632     \XKV@addtolist@x\XKV@naa\XKV@tempa
633   }%
```

Initialize the remaining keys, but only for the outermost level of `\setkeys`.

```
634   \ifnum\XKV@depth=\z@\let\XKV@rm\@empty\fi
```

Now scan the list of families for preset keys and set user input keys.

```

635   \XKV@usepresetkeys{#1}{preseth}%
636   \expandafter\XKV@s@tkeys\expandafter{\XKV@resb}{#1}%
637   \XKV@usepresetkeys{#1}{presett}%
638   \let\CurrentOption\@empty
639 }
```

`\XKV@usepresetkeys` `{⟨na⟩} {⟨postfix⟩}`

Loop over the list of families and check them for preset keys. If present, set them right away, taking into account the keys which are set by the user, available in the `\XKV@naa` list.

```

640 \def\XKV@usepresetkeys#1#2{%
641   \XKV@presettrue
642   \XKV@for@eo\XKV@fams\XKV@tfam{%
643     \XKV@makehd\XKV@tfam
644     \XKV@ifundefined{XKV@\XKV@header#2}{-}{%
645       \XKV@toks\expandafter\expandafter\expandafter
646         {\csname XKV@\XKV@header#2\endcsname}%
647       \@expandtwoargs\XKV@s@tkeys{the\XKV@toks}%
648       {\XKV@naa\ifx\XKV@naa\@empty\else,\fi#1}%
649     }%
650   }%
651   \XKV@presetfalse
652 }
```

```

\XKV@s@tkeys  {<key=value list>}{<na>}
This macro starts the loop over the key=value list. Do not set keys in the list <na>.
653 \def\XKV@s@tkeys#1#2{%
Define the list of key names which should be ignored.
654  \XKV@sp@deflist\XKV@na{#2}%
Loop over the key=value list.
655  \XKV@for@n{#1}\CurrentOption{%
Split key and value.
656    \expandafter\XKV@s@tk@ys\CurrentOption==\@nil
657  }%
658 }

\XKV@s@tk@ys  <key>=<value>=#3\@nil
Split key name and value (if present). If #3 non-empty, there was no =<value>.
659 \def\XKV@s@tk@ys#1=#2=#3\@nil{%
Check for \savevalue and \gsavevalue and remove spaces from around the key
name.
660  \XKV@g@tkeyname#1=\@nil\XKV@tkey
661  \expandafter\KV@@sp@def\expandafter\XKV@tkey\expandafter{\XKV@tkey}%
If the key is empty and a value has been specified, generate an error.
662  \ifx\XKV@tkey\@empty
663    \XKV@toks{#2}%
664    \ifcat$\the\XKV@toks$\else
665      \XKV@err{no key specified for value ‘\the\XKV@toks’}%
666    \fi
667  \else
If in the \XKV@na list, ignore the key.
668  \@expandtwoargs\in@{,\XKV@tkey,}{,\XKV@na,}%
669  \ifin@else
670    \XKV@knftrue
671    \KV@@sp@def\XKV@tempa{#2}%
672    \ifXKV@preset\XKV@s@tk@ys@{#3}\else
673      \ifXKV@pl
If a command with a + is used, set keys in all families on the list.
674      \XKV@for@eo\XKV@fams\XKV@tfam{%
675        \XKV@makehd\XKV@tfam
676        \XKV@s@tk@ys@{#3}%
677      }%
678      \else
Else, scan the families on the list but stop when the key is found or when the list has
run out.
679      \XKV@whilist\XKV@fams\XKV@tfam\ifXKV@knf\fi{%
680        \XKV@makehd\XKV@tfam
681        \XKV@s@tk@ys@{#3}%
682      }%
683      \fi
684      \fi
685      \ifXKV@knf
686      \ifXKV@inpox

```

We are in the options section. Try to use the macro defined by `\DeclareOptionX*`.

```
687 \ifx\XKV@doxs\relax
```

For classes, ignore unknown (possibly global) options. For packages, raise the standard  $\TeX$  error.

```
688 \ifx\@current\@clsextension\else
689 \let\CurrentOption\XKV@tkey\@unknownoptionerror
690 \fi
```

Pass the option through `\DeclareOptionX*`.

```
691 \else\XKV@doxs\fi
692 \else
```

If not in the options section, raise an error or add the key to the list in `\XKV@rm` when `\setkeys*` has been used.

```
693 \ifXKV@st
694 \XKV@addtolist@o\XKV@rm\CurrentOption
695 \else
696 \XKV@err{'\XKV@tkey' undefined in families '\XKV@fams'}%
697 \fi
698 \fi
699 \else
```

Remove global options set by the document class from `\@unusedoptionlist`. Global options set by other packages or classes will be removed by `\ProcessOptionsX*`.

```
700 \ifXKV@inpo\ifx\XKV@testclass\XKV@documentclass
701 \expandafter\XKV@useoption\expandafter{\CurrentOption}%
702 \fi\fi
703 \fi
704 \fi
705 \fi
706 }
```

```
\XKV@s@tk@ys@ {<ind>}
```

This macro coordinates the work of setting a key. `<ind>` is an indicator for the presence of a user submitted value for the key. If empty, no value was present.

```
707 \def\XKV@s@tk@ys@#1{%
```

Check whether the key macro exists.

```
708 \XKV@ifundefined{\XKV@header\XKV@tkey}{-}{%
709 \XKV@knfalse
```

Check global setting by `\savekeys` to know whether or not to save the value of the key at hand.

```
710 \XKV@ifundefined{XKV@\XKV@header save}{-}{%
711 \expandafter\XKV@testsavekey\csname XKV@\XKV@header
712 save\endcsname\XKV@tkey
713 }%
```

Save the value of a key.

```
714 \ifXKV@rkv
715 \ifXKV@sg\expandafter\global\fi\expandafter\let
716 \csname XKV@\XKV@header\XKV@tkey @value\endcsname\XKV@tempa
717 \fi
```

Replace pointers by saved values.

```
718 \expandafter\XKV@replacepointers\expandafter{\XKV@tempa}%
```

If no value was present, use the default value macro, if one exists. Otherwise, issue an error.

```

719 \ifx\@empty#1\@empty\XKV@afterelsefi
720 \XKV@ifundefined{\XKV@header\XKV@tkey @default}{%
721 \XKV@err{no value specified for key '\XKV@tkey'}%
722 }{%
723 \expandafter\expandafter\expandafter\XKV@default
724 \csname\XKV@header\XKV@tkey @default\endcsname\@nil
725 }%
726 \else\XKV@afterfi

```

Save state in case the key executes `\setkeys` or `\XKV@cc`.

```

727 \XKV@srstate{@\romannumeral\XKV@depth}{}%

```

Execute the key.

```

728 \csname\XKV@header\XKV@tkey\expandafter
729 \endcsname\expandafter{\XKV@tempa}\relax

```

Restore the current state.

```

730 \XKV@srstate{}{@\romannumeral\XKV@depth}%
731 \fi
732 }%
733 }

```

`\XKV@testsavekey` *<save key list>* *<key name>*

This macro checks whether the key in macro *<key name>* appears in the save list in macro *<save key list>*. Furthermore, it checks whether or not to save the key globally. In other words, that `\global{key}` is in the list.

```

734 \def\XKV@testsavekey#1#2{%
735 \ifXKV@rkv\else
736 \XKV@for@o#1\XKV@resa{%
737 \expandafter\XKV@ifcmd\expandafter{\XKV@resa}\global\XKV@resa{%
738 \ifx#2\XKV@resa
739 \XKV@rkvtrue\XKV@sgtrue
740 \fi
741 }{%
742 \ifx#2\XKV@resa
743 \XKV@rkvtrue\XKV@sgfalse
744 \fi
745 }%
746 }%
747 \fi
748 }

```

`\XKV@replacepointers` *{<key=value list>}*

`\XKV@r@placepointers` Replaces all pointers by their saved values. The result is stored in `\XKV@tempa`. We feed the replacement and the following tokens again to the macro to replace nested pointers. It stops when no pointers are found anymore. We keep a list of pointers replaced already for this key in `\XKV@resa` so we can check whether we are running in circles.

```

749 \def\XKV@replacepointers#1{%
750 \let\XKV@tempa\@empty
751 \let\XKV@resa\@empty
752 \XKV@r@placepointers#1\usevalue\@nil

```

```

753 }
754 \def\XKV@r@placepointers#1\usevalue#2{%
755   \XKV@addtomacro@n\XKV@tempa{#1}%
756   \def\XKV@tempb{#2}%
757   \ifx\XKV@tempb\@nnil\else\XKV@afterfi
758     \XKV@ifundefined{XKV@\XKV@header#2@value}{%
759       \XKV@err{no value recorded for key ‘#2’; ignored}%
760       \XKV@r@placepointers
761     }{%
762       \@expandtwoargs\in@{,#2,}{,\XKV@resa,}%
763       \ifin@\XKV@afterelsefi
764       \XKV@err{back linking pointers; pointer replacement canceled}%
765       \else\XKV@afterfi
766       \XKV@addtolist@x\XKV@resa{#2}%
767       \expandafter\expandafter\expandafter\XKV@r@placepointers
768       \csname XKV@\XKV@header#2@value\endcsname
769     \fi
770   }%
771 \fi
772 }

```

`\XKV@default` *<token>* *<tokens>*

This macro checks the `\prefix@fam@key@default` macro. If the macro has the form as defined by `keyval` or `xkeyval`, it is possible to extract the default value and safe that (if requested) and replace pointers. If the form is incorrect, just execute the macro and forget about possible pointers. The reason for this check is that certain packages (like `fancyvrb`) abuse the ‘default value system’ to execute code instead of setting keys by redefining default value macros. These macros do not actually contain a default value and trying to extract that would not work.

```
773 \def\XKV@default#1#2\@nil{%
```

Retrieve the first token in the macro.

```
774   \expandafter\edef\expandafter\XKV@tempa
775   \expandafter{\expandafter\@gobble\string#1}%
```

Construct the name that we expect on the basis of the `keyval` and `xkeyval` syntax of default values.

```
776   \edef\XKV@tempb{\XKV@header\XKV@tkey}%
```

Sanitize `\XKV@tempb` to reset catcodes for comparison with `\XKV@tempa`.

```
777   \@onelevel@sanitize\XKV@tempb
778   \ifx\XKV@tempa\XKV@tempb
```

If it is safe, extract the value. We temporarily redefine the key macro to save the default value in a macro. Saving the default value itself directly to a macro when defining keys would of course be easier, but a lot of packages rely on this system created by `keyval`, so we have to support it here.

```

779   \begingroup
780     \expandafter\def\csname\XKV@header\XKV@tkey\endcsname##1{%
781       \gdef\XKV@tempa{##1}%
782     }%
783     \csname\XKV@header\XKV@tkey @default\endcsname
784   \endgroup

```

Save the default value to a value macro if either the key name has been entered in a `\savekeys` macro or the starred form has been used.

```

785 \XKV@ifundefined{XKV@XKV@header save}{}{%
786 \expandafter\XKV@testsavekey\csname XKV@XKV@header
787 save\endcsname\XKV@tkey
788 }%
789 \ifXKV@rkv
790 \ifXKV@sg\expandafter\global\fi\expandafter\let
791 \csname XKV@XKV@header\XKV@tkey @value\endcsname\XKV@tempa
792 \fi

```

Replace the pointers.

```

793 \expandafter\XKV@replacepointers\expandafter
794 {\XKV@tempa}\XKV@afterelsefi

```

Save internal state.

```

795 \XKV@srstate{@\romannumeral\XKV@depth}{}%

```

Execute the key with the (possibly changed) default value.

```

796 \expandafter#1\expandafter{\XKV@tempa}\relax

```

Restore internal state.

```

797 \XKV@srstate{ }{@\romannumeral\XKV@depth}%
798 \else\XKV@afterfi

```

Save internal state.

```

799 \XKV@srstate{@\romannumeral\XKV@depth}{}%

```

Execute the key with the default value.

```

800 \csname\XKV@header\XKV@tkey @default\endcsname\relax

```

Restore the state.

```

801 \XKV@srstate{ }{@\romannumeral\XKV@depth}%
802 \fi
803 }

```

`\setrmkeys` *\*+ [*prefix*] {*families*}*

Set remaining keys stored in `\XKV@rm`. The starred version creates a new list in `\XKV@rm` in case there are still keys that cannot be located in the families specified. Care is taken again not to expand fragile macros. Use `\XKV@testopa` again to handle optional arguments.

```

804 \def\setrmkeys{\XKV@testopta{\XKV@testoptc\XKV@setrmkeys}}

```

`\XKV@setrmkeys` [*na*]

Submits the keys in `\XKV@rm` to `\XKV@setkeys`.

```

805 \def\XKV@setrmkeys[#1]{%
806 \def\XKV@tempa{\XKV@setkeys[#1]}%
807 \expandafter\XKV@tempa\expandafter{\XKV@rm}%
808 }

```

Reset catcodes.

```

809 \XKV@catcodes
810 </xkvtext>

```

## 14.2 xkeyval.sty

Initialize the package.

```
811 < *xkvlatex >
812 \NeedsTeXFormat{LaTeX2e}[1995/12/01]
813 \ProvidesPackage{xkeyval}
814 [2008/08/13 v2.6a package option processing (HA)]
```

Initializations. Load xkeyval.tex, adjust some catcodes to define internal macros and initialize the \DeclareOptionX\* working macro.

```
815 \ifx\XKeyValLoaded\endinput\else\input xkeyval \fi
816 \edef\XKVcatcodes{%
817   \catcode'\noexpand\=\the\catcode'\=\relax
818   \catcode'\noexpand\,\the\catcode'\,\relax
819   \let\noexpand\XKVcatcodes\relax
820 }
821 \catcode'\=12\relax
822 \catcode'\,12\relax
823 \let\XKV@doxs\relax
```

\XKV@warn Warning and error macros.

```
\XKV@err 824 \def\XKV@warn#1{\PackageWarning{xkeyval}{#1}}
825 \def\XKV@err#1{\PackageError{xkeyval}{#1}\@ehc}
```

Retrieve the document class from \@filelist. This is the first filename in the list with a class extension. Use a while loop to scan the list and stop when we found the first filename which is a class. Also stop in case the list is scanned fully.

```
826 \XKV@whilst\@filelist\XKV@tempa\ifx\XKV@documentclass\undefined\fi{%
827   \filename@parse\XKV@tempa
828   \ifx\filename@ext\clsextension
829     \XKV@ifundefined{opt@\filename@area\filename@base.\filename@ext}
830     }-}{-%
831     \edef\XKV@documentclass{%
832       \filename@area\filename@base.\filename@ext
833     }%
834   }%
835 \fi
836 }
```

If we didn't find the document class, raise an error, otherwise filter global options.

```
837 \ifx\XKV@documentclass\undefined
838   \XKV@err{xkeyval loaded before \protect\documentclass}%
839   \let\XKV@documentclass\@empty
840   \let\XKV@classoptionslist\@empty
841 \else
842   \let\XKV@classoptionslist\@classoptionslist
```

Code to filter key=value pairs from \@classoptionslist without expanding options.

```
843 \def\XKV@tempa#1{%
844   \let\@classoptionslist\@empty
845   \XKV@for@n{#1}\XKV@tempa{%
846     \expandafter\in\expandafter=\expandafter{\XKV@tempa}%
847     \ifin\else\XKV@addtolist\@o\@classoptionslist\XKV@tempa\fi
848   }%
```

```

849 }
850 \expandafter\XKV@tempa\expandafter{\@classoptionslist}
851 \fi

```

```

\XKV@testopte  {\langle function \rangle}
\XKV@t@stopte  Macros for \ExecuteOptionsX and \ProcessOptionsX for testing for optional argu-
\XKV@t@st@pte  ments and inserting default values. Execute \langle function \rangle after preforming the checks.
\XKV@@@t@st@pte 852 \def\XKV@testopte#1{%
853   \XKV@ifstar{\XKV@sttrue\XKV@t@stopte#1}{\XKV@stfalse\XKV@t@stopte#1}%
854 }
855 \def\XKV@t@stopte#1{\@testopt{\XKV@t@st@pte#1}{KV}}
856 \def\XKV@t@st@pte#1[#2]{%
857   \XKV@makepf{#2}%
858   \@ifnextchar<{\XKV@@@t@st@pte#1}%
859   {\XKV@@@t@st@pte#1<\@currname.\@current>}%
860 }
861 \def\XKV@@@t@st@pte#1<#2>{%
862   \XKV@sp@deflist\XKV@fams{#2}%
863   \@testopt#1{}}
864 }

```

Macros for class and package writers. These are mainly shortcuts to `\define@key` and `\setkeys`. The  $\TeX$  macro `\@fileswith@ptions` is set to generate an error. This is the case when a class or package is loaded in between `\DeclareOptionX` and `\ProcessOptionsX` commands.

```

\DeclareOptionX *
  Declare a package or class option.
865 \def\DeclareOptionX{%
866   \let\@fileswith@ptions\@badrequireerror
867   \XKV@ifstar\XKV@dox\XKV@d@x
868 }

```

```

\XKV@dox  This macro defines \XKV@doxs to be used for unknown options.
869 \long\def\XKV@dox#1{\XKV@toks{#1}\edef\XKV@doxs{\the\XKV@toks}}

```

```

\XKV@d@x  Insert default prefix and family name (which is the filename of the class or package)
\XKV@@@d@x and add empty default value if none present. Execute \define@key.
\XKV@@@@d@x 870 \def\XKV@d@x{\@testopt\XKV@@@d@x{KV}}
871 \def\XKV@@@@d@x[#1]{%
872   \@ifnextchar<{\XKV@@@@d@x[#1]}{\XKV@@@@d@x[#1]<\@currname.\@current>}%
873 }
874 \def\XKV@@@@d@x[#1]<#2>#3{\@testopt{\define@key[#1]{#2}{#3}}{}}

```

```

\ExecuteOptionsX  [\langle prefix \rangle]{\langle families \rangle}[\langle na \rangle]{\langle key=value list \rangle}
  This macro sets keys to specified values and uses \XKV@setkeys to do the job. In-
  sert default prefix and family name if none provided. Use \XKV@t@stopte to handle
  optional arguments and reset \ifXKV@st and \ifXKV@pl first to avoid unexpected
  behavior when \setkeys*+ (or a friend) has been used before \ExecuteOptionsX.
875 \def\ExecuteOptionsX{\XKV@stfalse\XKV@plfalse\XKV@t@stopte\XKV@setkeys}

```



```

\ProcessOptionsX *[\<prefix>]{\<families>}
Processes class or package using xkeyval. The starred version copies class options sub-
mitted by the user as well, given that they are defined in the local families which are
passed to the macro. Use \XKV@testopte to handle optional arguments.
876 \def\ProcessOptionsX{\XKV@plfalse\XKV@testopte\XKV@pox}

\XKV@pox [\<na>]
Workhorse for \ProcessOptionsX and \ProcessOptionsX*.
877 \def\XKV@pox[#1]{%
878 \let\XKV@tempa\@empty

Set \XKV@inpox: indicates that we are in \ProcessOptionsX to invoke a special rou-
tine in \XKV@s@tkeys.
879 \XKV@inpoxtrue

Set \@fileswith@pti@ns again in case no \DeclareOptionX has been used. This
will be used to identify a call to \setkeys from \ProcessOptionsX.
880 \let\@fileswith@pti@ns\@badrequireerror
881 \edef\XKV@testclass{\@currname.\@currentx}%

If xkeyval is loaded by the document class, initialize \@unusedoptionlist.
882 \ifx\XKV@testclass\XKV@documentclass
883 \let\@unusedoptionlist\XKV@classoptionslist
884 \XKV@ifundefined{ver@xkvltxp.sty}{-}{%
885 \@onelevel@sanitize\@unusedoptionlist
886 }%
887 \else

Else, if the starred version is used, copy global options in case they are defined in local
families. Do not execute this in the document class to avoid setting keys twice.
888 \ifXKV@st
889 \def\XKV@tempb##1,{%
890 \def\CurrentOption{##1}%
891 \ifx\CurrentOption\@nnil\else
892 \XKV@g@tkeyname##1=\@nil\CurrentOption
893 \XKV@key@if@defined{\CurrentOption}{-}{%

If the option also exists in local families, add it to the list for later use and remove it
from \@unusedoptionlist.
894 \XKV@useoption{##1}%
895 \XKV@addtolist@n\XKV@tempa{##1}%
896 }%
897 \expandafter\XKV@tempb
898 \fi
899 }%
900 \expandafter\XKV@tempb\XKV@classoptionslist,\@nil,%
901 \fi
902 \fi

Add current package options to the list.
903 \expandafter\XKV@addtolist@o\expandafter
904 \XKV@tempa\csname opt@\@currname.\@currentx\endcsname

Set options. We can be certain that global options can be set since the definitions of lo-
cal options have been checked above. Note that \DeclareOptionX* will not consume
global options when \ProcessOptionsX* is used.

```

```

905 \def\XKV@tempb{\XKV@setkeys[#1]}%
906 \expandafter\XKV@tempb\expandafter{\XKV@tempa}%
Reset the macro created by \DeclareOptionX* to avoid processing future unknown
keys using \XKV@doxs.
907 \let\XKV@doxs\relax
Reset the \XKV@rm macro to avoid processing remaining options with \setrmkeys.
908 \let\XKV@rm\@empty
Reset \ifXKV@inpox: not in \ProcessOptionsX anymore.
909 \XKV@inpoxfalse
Reset \@fileswith@pti@ns to allow loading of classes or packages again.
910 \let\@fileswith@pti@ns\@@fileswith@pti@ns
911 \AtEndOfPackage{\let\@unprocessedoptions\relax}%
912 }

```

```

\XKV@useoption {<option>}
Removes an option from \@unusedoptionlist.
913 \def\XKV@useoption#1{%
914 \def\XKV@resa{#1}%
915 \XKV@ifundefined{ver@xkvltxp.sty}{-}{%
916 \@onelevel@sanitize\XKV@resa
917 }%
918 \@expandtwoargs\@removeelement{\XKV@resa}%
919 {\@unusedoptionlist}\@unusedoptionlist
920 }

```

The options section. Postponed to the end to allow for using xkeyval options macros. All options are silently ignored.

```

921 \DeclareOptionX*{%
922 \PackageWarning{xkeyval}{Unknown option ‘\CurrentOption’}%
923 }
924 \ProcessOptionsX
Reset catcodes.
925 \XKV@catcodes
926 </xkvltx>

```

### 14.3 keyval.tex

Since the xkeyval macros handle input in a very different way than keyval macros, it is not wise to redefine keyval primitives (like \KV@do and \KV@split) used by other packages as a back door into \setkeys. Instead, we load the original primitives here for compatibility to existing packages using (parts of) keyval. Most of the code is original, but slightly adapted to xkeyval. See the keyval documentation for information about the macros below.

```

927 (*xkvkeyval)
928 %%
929 %% Based on keyval.sty.
930 %%
931 \def\XKV@tempa#1{%
932 \def\KV@@sp@def##1##2{%

```

```

933 \futurelet\XKV@resa\KV@@sp@d##2\@nil\@nil#1\@nil\relax##1}%
934 \def\KV@@sp@d{%
935 \ifx\XKV@resa\@sptoken
936 \expandafter\KV@@sp@b
937 \else
938 \expandafter\KV@@sp@b\expandafter#1%
939 \fi}%
940 \def\KV@@sp@b#1##1 \@nil{\KV@@sp@c##1}%
941 }
942 \XKV@tempa{ }
943 \def\KV@@sp@c#1\@nil#2\relax#3{\XKV@toks{#1}\edef#3{\the\XKV@toks}}
944 \def\KV@do#1,{%
945 \ifx\relax#1\@empty\else
946 \KV@split#1==\relax
947 \expandafter\KV@do\fi}
948 \def\KV@split#1=#2=#3\relax{%
949 \KV@@sp@def\XKV@tempa{#1}%
950 \ifx\XKV@tempa\@empty\else
951 \expandafter\let\expandafter\XKV@tempc
952 \csname\KV@prefix\XKV@tempa\endcsname
953 \ifx\XKV@tempc\relax
954 \XKV@err{'\XKV@tempa' undefined}%
955 \else
956 \ifx\@empty#3\@empty
957 \KV@default
958 \else
959 \KV@@sp@def\XKV@tempb{#2}%
960 \expandafter\XKV@tempc\expandafter{\XKV@tempb}\relax
961 \fi
962 \fi
963 \fi}
964 \def\KV@default{%
965 \expandafter\let\expandafter\XKV@tempb
966 \csname\KV@prefix\XKV@tempa @default\endcsname
967 \ifx\XKV@tempb\relax
968 \XKV@err{No value specified for key '\XKV@tempa'}%
969 \else
970 \XKV@tempb\relax
971 \fi}
972 \def\KV@def#1#2[#3]{%
973 \@namedef{KV@#1@#2@default\expandafter}\expandafter
974 {\csname KV@#1@#2\endcsname{#3}}%
975 \@namedef{KV@#1@#2}##1}
976 </xkvkeyval>

```

#### 14.4 xkvtxhdr.tex

This section generates `xkvtxhdr.tex` which contains some standard  $\text{\LaTeX}$  macros taken from `latex.ltx`. This will only be loaded when not using `xkeyval.sty`.

```

977 <*xkvheader>
978 %%
979 %% Taken from latex.ltx.
980 %%

```

```

981 \message{2005/02/22 v1.1 xkeyval TeX header (HA)}
982 \def\@nnil{\@nil}
983 \def\@empty{}
984 \def\newif#1{%
985   \count@\escapechar \escapechar\m@ne
986   \let#1\iffalse
987   \@if#1\iftrue
988   \@if#1\iffalse
989   \escapechar\count@}
990 \def\@if#1#2{%
991   \expandafter\def\csname\expandafter\@gobbletwo\string#1%
992     \expandafter\@gobbletwo\string#2\endcsname
993     {\let#1#2}}
994 \long\def\@ifnextchar#1#2#3{%
995   \let\reserved@d=#1%
996   \def\reserved@a{#2}%
997   \def\reserved@b{#3}%
998   \futurelet\@let@token\@ifnch}
999 \def\@ifnch{%
1000  \ifx\@let@token\@sptoken
1001    \let\reserved@c\@xifnch
1002  \else
1003    \ifx\@let@token\reserved@d
1004      \let\reserved@c\reserved@a
1005    \else
1006      \let\reserved@c\reserved@b
1007    \fi
1008  \fi
1009  \reserved@c}
1010 \def\:{\let\@sptoken= } \: % this makes \@sptoken a space token
1011 \def\:{\@xifnch} \expandafter\def\:{ {\futurelet\@let@token\@ifnch}
1012 \let\kernel@ifnextchar\@ifnextchar
1013 \long\def\@testopt#1#2{%
1014   \kernel@ifnextchar[#{1}{#1[#{2}]}]}
1015 \long\def\@firstofone#1{#1}
1016 \long\def \@gobble #1{}
1017 \long\def \@gobbletwo #1#2{}
1018 \def\@expandtwoargs#1#2#3{%
1019 \edef\reserved@a{\noexpand#1{#2}{#3}}\reserved@a}
1020 \edef\@backslashchar{\expandafter\@gobble\string\}
1021 \newif\ifin@
1022 \def\in@#1#2{%
1023   \def\in@@##1##2##3\in@@{%
1024     \ifx\in@@#2\in@false\else\in@true\fi}%
1025   \in@@#2#1\in@\in@@}
1026 \def\strip@prefix#1>{}
1027 \def \@onelevel@sanitize #1{%
1028   \edef #1{\expandafter\strip@prefix
1029     \meaning #1}%
1030 }
1031 </xkvheader>

```

## 14.5 xkvview.sty

This section provides a small utility for package developers. It provides several macros to generate overviews of the keys that are defined in a package or a collection of packages. It is possible to get an overview for a specific family, but also to get a complete overview of all keys that have been defined after loading this package.

```
1032 (*xkvview)
1033 \NeedsTeXFormat{LaTeX2e}[1995/12/01]
1034 \ProvidesPackage{xkvview}%
1035 [2008/08/10 v1.5 viewer utility for xkeyval (HA)]
1036 \RequirePackage{xkeyval}
1037 \RequirePackage{longtable}
1038 \DeclareOptionX*{%
1039   \PackageWarning{xkvview}{Unknown option ‘\CurrentOption’}%
1040 }
1041 \ProcessOptionsX

  Initializations.

1042 \newif\ifXKV@vwkey
1043 \newif\ifXKV@colii
1044 \newif\ifXKV@coliii
1045 \newif\ifXKV@coliv
1046 \newif\ifXKV@colv
1047 \newwrite\XKV@out
1048 \let\XKV@db\@empty

  Setup options and presets.

1049 \define@cmdkeys[XKV]{xkvview}[XKV@]{%
1050   prefix,family,type,default,file,columns,wcolsep,weol}[\@nil]
1051 \define@boolkeys[XKV]{xkvview}[XKV@]{view,vlabels,wlabels}[true]
1052 \presetkeys[XKV]{xkvview}{prefix,family,type,default,file,%
1053   columns,wcolsep=&,weol=\\,view,vlabels=false,wlabels=false}{%

\XKV@tabulate {<key>}{<type>}{<default>}
\XKV@t@bulate Adds the input information to the main database in \XKV@db.

1054 \def\XKV@tabulate#1#2#3{%
1055   \def\XKV@tempa{#3}%
1056   \@onelevel@sanitize\XKV@tempa
1057   \XKV@addtolist@x\XKV@db{#1=\ifx\XKV@prefix\@empty\else\expandafter
1058     \XKV@t@bulate\XKV@prefix\fi=\XKV@tfam=#2=\XKV@tempa}%
1059 }
1060 \def\XKV@t@bulate#1@{#1}

\XKV@define@key Redefine the internals of key defining macros to record information in the database.
\XKV@d@f@i@n@e@k@y 1061 \def\XKV@define@key#1{%
\XKV@define@cmdkey 1062   \@ifnextchar[{\XKV@d@f@i@n@e@k@y{#1}}]{%
\XKV@d@f@i@n@e@ch@i@c@e@y 1063     \XKV@tabulate{#1}{ordinary}{[none]}%
\XKV@d@f@i@n@e@ch@i@c@e@y 1064     \expandafter\def\csname\XKV@header#1\endcsname###1%
\XKV@d@f@i@n@e@b@o@l@k@e@y 1065     }%
1066 }
1067 \def\XKV@d@f@i@n@e@k@y#1[#2]{%
1068   \XKV@tabulate{#1}{ordinary}{#2}%
1069   \XKV@define@default{#1}{#2}%
1070   \expandafter\def\csname\XKV@header#1\endcsname##1%
```

```

1071 }
1072 \def\XKV@define@cmdkey#1#2[#3]#4{%
1073   \ifXKV@st
1074     \XKVV@tabulate{#2}{command}{#3}%
1075     \XKV@define@default{#2}{#3}%
1076   \else
1077     \XKVV@tabulate{#2}{command}{[none]}%
1078   \fi
1079 \def\XKV@tempa{\expandafter\def\csname\XKV@header#2\endcsname###1}%
1080 \begingroup\expandafter\endgroup\expandafter\XKV@tempa\expandafter
1081   {\expandafter\def\csname#1#2\endcsname{##1}#4}%
1082 }
1083 \def\XKV@d@f@i@n@e@ch@i@k@e@y#1[#2]{%
1084   \XKVV@tabulate{#1}{choice}{#2}%
1085   \XKV@define@default{#1}{#2}%
1086   \XKV@d@f@i@n@e@ch@i@k@e@y{#1}%
1087 }
1088 \def\XKV@d@f@i@n@e@ch@i@k@e@y#1{%
1089   \XKVV@tabulate{#1}{choice}{[none]}%
1090   \ifXKV@pl\XKV@afterelsefi
1091     \expandafter\XKV@d@f@i@n@e@ch@i@k@e@y
1092   \else\XKV@afterfi
1093     \expandafter\XKV@d@f@i@n@e@ch@i@k@e@y
1094   \fi
1095   \csname\XKV@header#1\endcsname
1096 }
1097 \def\XKV@d@f@i@n@e@b@o@l@k@e@y#1#2#3#4#5{%
1098   \expandafter\newif\csname if#3\endcsname
1099   \ifXKV@st
1100     \XKVV@tabulate{#2}{boolean}{#4}%
1101     \XKV@define@default{#2}{#4}%
1102   \else
1103     \XKVV@tabulate{#2}{boolean}{[none]}%
1104   \fi
1105   \ifXKV@pl
1106     \def###1{\XKV@pltrue\XKV@sttrue
1107       \XKV@checkchoice[\XKV@resa]{##1}{true,false}#5%
1108     }%
1109   \else
1110     \def###1{\XKV@plfalse\XKV@sttrue
1111       \XKV@checkchoice[\XKV@resa]{##1}{true,false}#5%
1112     }%
1113   \fi
1114 }

```

`\xkvview` *{<options>}*

The main macro. Produces a long table and/or writes to a target file.

```

1115 \def\xkvview#1{%
  Process all options.
1116   \setkeys[XKVV]{xkvview}{#1}%
1117   \ifx\XKVV@default\@nnil\else\@onelevel@sanitize\XKVV@default\fi
  If no column information, display all columns.
1118   \ifx\XKVV@columns\@nnil

```

```

1119 \count@5
1120 \XKV@colii@true\XKV@coliii@true\XKV@coliv@true\XKV@colv@true
1121 \else

```

Check how much and which columns should be displayed.

```

1122 \count@\@ne
1123 \@expandtwoargs\in@{,prefix,}{, \XKV@columns,}%
1124 \ifin@\advance\count@\@ne\XKV@colii@true\else\XKV@colii@false\fi
1125 \@expandtwoargs\in@{,family,}{, \XKV@columns,}%
1126 \ifin@\advance\count@\@ne\XKV@coliii@true\else\XKV@coliii@false\fi
1127 \@expandtwoargs\in@{,type,}{, \XKV@columns,}%
1128 \ifin@\advance\count@\@ne\XKV@coliv@true\else\XKV@coliv@false\fi
1129 \@expandtwoargs\in@{,default,}{, \XKV@columns,}%
1130 \ifin@\advance\count@\@ne\XKV@colv@true\else\XKV@colv@false\fi
1131 \fi
1132 \ifXKV@view

```

Construct long table header.

```

1133 \protected@edef\XKV@tempa{\noexpand\begin{longtable}[l]{%
1134 * \the\count@ l}\normalfont Key\ifXKV@colii&\normalfont Prefix%
1135 \fi\ifXKV@coliii&\normalfont Family\fi\ifXKV@coliv&\normalfont
1136 Type\fi\ifXKV@colv&\normalfont Default\fi\\noexpand\hline
1137 \noexpand\endfirsthead\noexpand\multicolumn{\the\count@}{l}{%
1138 \normalfont\emph{Continued from previous page}}\\noexpand\hline
1139 \normalfont Key\ifXKV@colii&\normalfont Prefix\fi\ifXKV@coliii
1140 &\normalfont Family\fi\ifXKV@coliv&\normalfont Type\fi
1141 \ifXKV@colv&\normalfont Default\fi\\noexpand\hline\noexpand
1142 \endhead\noexpand\hline\noexpand\multicolumn{\the\count@}{r}{%
1143 \normalfont\emph{Continued on next page}}\\noexpand\endfoot
1144 \noexpand\hline\noexpand\endlastfoot
1145 }%
1146 \XKV@toks\expandafter{\XKV@tempa}%
1147 \fi

```

Open the target file for writing if a file name has been specified.

```

1148 \ifx\XKV@file\@nnil\else\immediate\openout\XKV@out\XKV@file\fi

```

Parse the entire database to find entries that match the criteria.

```

1149 \XKV@for@o\XKV@db\XKV@tempa{%
1150 \XKV@vwkey@true\expandafter\XKV@xkvview\XKV@tempa\@nil
1151 }%

```

Finish the long table and typeset it.

```

1152 \ifXKV@view
1153 \addto@hook\XKV@toks{\end{longtable}}%
1154 \begingroup\ttfamily\the\XKV@toks\endgroup
1155 \fi

```

Close the target file.

```

1156 \ifx\XKV@file\@nnil\else\immediate\closeout\XKV@out\fi
1157 }

```

```
\XKV@xkvview <key>=<prefix>=<family>=<type>=<default>\@nil
```

Parse a row in the database to get individual column entries. Select the requested columns and store the table row in the token or write it to the target file.

```
1158 \def\XKV@xkvview#1=#2=#3=#4=#5\@nil{%
```

Check whether the current entry satisfies all criteria.

```

1159 \ifx\XKV@prefix\@nnil\else
1160 \def\XKV@tempa{#2}%
1161 \ifx\XKV@tempa\XKV@prefix\else\XKV@vwkeyfalse\fi
1162 \fi
1163 \ifx\XKV@family\@nnil\else
1164 \def\XKV@tempa{#3}%
1165 \ifx\XKV@tempa\XKV@family\else\XKV@vwkeyfalse\fi
1166 \fi
1167 \ifx\XKV@type\@nnil\else
1168 \def\XKV@tempa{#4}%
1169 \ifx\XKV@tempa\XKV@type\else\XKV@vwkeyfalse\fi
1170 \fi
1171 \ifx\XKV@default\@nnil\else
1172 \def\XKV@tempa{#5}%
1173 \ifx\XKV@tempa\XKV@default\else\XKV@vwkeyfalse\fi
1174 \fi
1175 \ifXKV@vwkey

```

If output should go to the dvi, construct the table row and add it to the token.

```

1176 \ifXKV@view
1177 \edef\XKV@tempa{%
1178 #1\ifXKV@colii&#2\fi\ifXKV@coliii&#3\fi
1179 \ifXKV@coliv&#4\fi\ifXKV@colv&#5\fi
1180 \ifXKV@vlabels\noexpand\label{#2-#3-#1}\fi
1181 }%
1182 \expandafter\addto@hook\expandafter
1183 \XKV@toks\expandafter{\XKV@tempa\}%
1184 \fi
1185 \ifx\XKV@file\@nnil\else

```

When writing, construct the line and write it to file. Notice that `xkeyval` removes braces and spaces, so `wcolsep={ }` won't make a space between column entries, but `wcolsep=\space` will.

```

1186 \immediate\write\XKV@out{%
1187 #1\ifXKV@colii\XKV@wcolsep#2\fi
1188 \ifXKV@coliii\XKV@wcolsep#3\fi
1189 \ifXKV@coliv\XKV@wcolsep#4\fi
1190 \ifXKV@colv\XKV@wcolsep#5\fi
1191 \ifXKV@wlabels\string\label{#2-#3-#1}\fi
1192 \expandafter\noexpand\XKV@weol
1193 }%
1194 \fi
1195 \fi
1196 }
1197 </xkvview>

```

## 14.6 xkvltxp.sty

This section redefines some kernel macros as to avoid expansions of options at several places to allow for macros in key values in class and package options. It uses a temporary token register and some careful expansions. Notice that `\@unusedoptionlist` is sanitized after creation by `xkeyval` to avoid `\@removeelement` causing problems with



macros and braces. See for more information about the original versions of the macros below the kernel source documentation [2].

```

1198 (*xkvltxpatch)
1199 %%
1200 %% Based on latex.ltx.
1201 %%
1202 \NeedsTeXFormat{LaTeX2e}[1995/12/01]
1203 \ProvidesPackage{xkvltxp}[2004/12/13 v1.2 LaTeX2e kernel patch (HA)]
1204 \def\@pass@options#1#2#3{%
1205   \def\reserved@a{#2}%
1206   \def\reserved@b{\CurrentOption}%
1207   \ifx\reserved@a\reserved@b
1208     \@ifundefined{opt@#3.#1}{\@temptokena\expandafter{#2}}{%
1209       \@temptokena\expandafter\expandafter\expandafter
1210         {\csname opt@#3.#1\endcsname}%
1211       \@temptokena\expandafter\expandafter\expandafter{%
1212         \expandafter\the\expandafter\@temptokena\expandafter,#2}%
1213     }%
1214   \else
1215     \@ifundefined{opt@#3.#1}{\@temptokena{#2}}{%
1216       \@temptokena\expandafter\expandafter\expandafter
1217         {\csname opt@#3.#1\endcsname}%
1218       \@temptokena\expandafter{\the\@temptokena,#2}%
1219     }%
1220   \fi
1221   \expandafter\xdef\csname opt@#3.#1\endcsname{\the\@temptokena}%
1222 }
1223 \def\OptionNotUsed{%
1224   \ifx\@current\@clsextension
1225     \let\reserved@a\CurrentOption
1226     \@onelevel@sanitize\reserved@a
1227     \xdef\@unusedoptionlist{%
1228       \ifx\@unusedoptionlist\@empty\else\@unusedoptionlist,\fi
1229       \reserved@a}%
1230   \fi
1231 }
1232 \def\@use@option{%
1233   \let\reserved@a\CurrentOption
1234   \@onelevel@sanitize\reserved@a
1235   \@expandtwoargs\@removeelement\reserved@a
1236   \@unusedoptionlist\@unusedoptionlist
1237   \csname ds@\CurrentOption\endcsname
1238 }
1239 \def\@fileswith@ptions#1[#2]#3[#4]{%
1240   \ifx#1\@clsextension
1241     \ifx\@classoptionslist\relax
1242       \@temptokena{#2}%
1243       \xdef\@classoptionslist{\the\@temptokena}%
1244       \def\reserved@a{%
1245         \@onefilewithoptions#3[#2][#4]#1%
1246         \@documentclasshook}%
1247     \else
1248       \def\reserved@a{%
1249         \@onefilewithoptions#3[#2][#4]#1}%

```

```

1250 \fi
1251 \else
1252 \@temptokena{#2}%
1253 \def\reserved@b##1,{%
1254 \ifx\@nil##1\relax\else
1255 \ifx\relax##1\relax\else
1256 \noexpand\@onefilewithoptions##1%
1257 [\the\@temptokena][#4]\noexpand\@pkgextension
1258 \fi
1259 \expandafter\reserved@b
1260 \fi}%
1261 \edef\reserved@a{\zap@space#3 \@empty}%
1262 \edef\reserved@a{\expandafter\reserved@b\reserved@a,\@nil,}%
1263 \fi
1264 \reserved@a}
1265 \let\@fileswith@pti@ns\@fileswith@pti@ns
1266 </xkv\txpatch>

```

## 14.7 pst-xkey.tex

Avoid loading `pst-xkey.tex` twice.

```

1267 (*pxktx)
1268 \csname PSTXKeyLoaded\endcsname
1269 \let\PSTXKeyLoaded\endinput
1270 \edef\PSTXKeyCatcodes{%
1271 \catcode'\noexpand\@the\catcode'\@relax
1272 \let\noexpand\PSTXKeyCatcodes\relax
1273 }
1274 \catcode'\@=11\relax

```

Load `xkeyval` when not already done by `pst-xkey.sty` and provide information.

```

1275 \ifx\ProvidesFile\@undefined
1276 \message{2005/11/25 v1.6 PSTricks specialization of xkeyval (HA)}
1277 \ifx\XKeyValLoaded\endinput\else\input xkeyval \fi
1278 \else
1279 \ProvidesFile{pst-xkey.tex}
1280 [2005/11/25 v1.6 PSTricks specialization of xkeyval (HA)]
1281 \@addtofilelist{pst-xkey.tex}
1282 \RequirePackage{xkeyval}
1283 \fi

```

`\pst@famlist` Initialize the list of families.

```
1284 \def\pst@famlist{}
```

`\pst@addfams` Adds the family to `\pst@famlist` if it was not in yet.

```

1285 \def\pst@addfams#1{%
1286 \XKV@for@n{#1}\XKV@tempa{%
1287 \@expandtwoargs\in@{\XKV@tempa,}\pst@famlist,}%
1288 \ifin@\else\edef\pst@famlist{\pst@famlist,\XKV@tempa}\fi
1289 }%
1290 }

```

`\psset` Set keys. Uses `xkeyval's \setkeys+`.

`\pss@t`



## Acknowledgements

The author is grateful to Carsten Heinz, Morten Høgholm, Josselin Noirel, Heiko Oberdiek, Will Robertson, Till Tantau, Herbert Voß and Joseph Wright for help and suggestions. Thanks go to Donald Arseneau for contributing the robust `\@ifnextcharacter` macro and to Morten Høgholm for contributing a fast for-loop macro. Special thanks go to Uwe Kern for his ideas for improving the functionality of this package, a lot of useful comments on the package and the documentation and for contributing the `\@selective@sanitize` macro.

## Version history

This version history displays recent changes only.

<b>v2.0</b>	<i>(2005/01/30)</i>
General: Made <code>\setkeys</code> nestable	1
<code>\XKV@addtolist@n</code> : Simplified	32
<code>\XKV@addtolist@o</code> : Simplified	32
<code>\XKV@default</code> : Repaired adding extra braces when executing default value	53
<code>\XKV@ifundefined</code> : Made none $\epsilon$ -TeX version not leave <code>\relax</code>	29
<code>\XKV@r@placepointers</code> : Simplified	52
<b>v2.1</b>	<i>(2005/02/08)</i>
General: Added ‘immediate’ versions of several macros	1
<b>v2.2</b>	<i>(2005/02/14)</i>
General: Added viewer utility	1
Improved nesting mechanism	1
<b>v2.3</b>	<i>(2005/02/22)</i>
General: Added choice keys	1
Increased efficiency of loops	1
Updated viewer utility	1
<b>v2.4</b>	<i>(2005/03/31)</i>
General: Added ‘default value’ column to <code>xkvview</code> tables	1
Added nesting protection for conditionals	1
Changed <code>\define@boolkey</code> to have a key function	1
Extended boolean keys	1
Extended choice keys	1
Inserted <code>pst-xkey</code> in <code>xkeyval</code> source	1
Removed command keys	1
Revised documentation and examples	1
Simplified some code	1
Updated <code>xkvview</code>	1
<code>\XKV@s@tk@ys</code> : Added <code>\global</code> to make <code>\XKV@rm</code> survive when <code>\setkeys</code> executed in a group.	50
<code>\XKV@wh@list</code> : Avoid using grouping	31
<b>v2.5</b>	<i>(2005/05/07)</i>
General: Added <code>\define@boolkeys</code> , <code>\define@cmdkey</code> and <code>\define@cmdkeys</code>	1
Restructured documentation	1
Simplified <code>\setkeys</code> internals	1
Solved small bug in <code>\setkeys</code> which allowed other families to take over save key or preset key settings if the key was defined in multiple families	1
Updated <code>xkvview</code>	1
<code>\XKV@d@f@ne@boolkey</code> : Removed <code>\relax</code>	42

\XKV@d@fine@boolkey: Removed \relax .....	42
<b>v2.5</b>	<i>(2005/05/21)</i>
General: Added default value examples to docs .....	1
Reimplemented xkvview and added several options .....	1
<b>v2.5a</b>	<i>(2005/05/31)</i>
@s@lective@sanitize: Added missing '%' .....	32
<b>v2.5b</b>	<i>(2005/06/20)</i>
General: Made retrieving document class more robust .....	1
<b>v2.5c</b>	<i>(2005/07/10)</i>
\XKV@define@cmdkey: Avoid initializing control sequence as \relax .....	40, 61
<b>v2.5d</b>	<i>(2005/08/12)</i>
General: Added missing \filename@area in document class retrieval in xkeyval.sty .	1
<b>v2.5e</b>	<i>(2005/11/25)</i>
General: Updated docs .....	1
\psset: Added \ignorespaces as in pstricks.tex .....	66
<b>v2.5f</b>	<i>(2006/11/18)</i>
\XKV@setkeys: Added reset of \CurrentOption .....	49
\XKV@srstate: Added XKV@tkey and XKV@rm to solve bugs .....	37
<b>v2.5g</b>	<i>(2006/12/19)</i>
General: Altered policy for handling \XKV@rm in nested \setkeys* commands: all un-	
known keys will be recorded, not only the once from the outermost \setkeys* .	1
\XKV@s@tk@ys: Removed \global again for consistent approach of \XKV@rm and to al-	
low groups to keep \XKV@rm local .....	50
\XKV@setkeys: Avoid reset of \XKV@rm in nested \setkeys commands .....	49
\XKV@srstate: Removed XKV@rm again on user request .....	37
<b>v2.6</b>	<i>(2008/08/10)</i>
@s@lective@sanitize: protecting assignments from # .....	32
General: Added \KV@def to keyval.tex as it is used by some packages .....	58
\XKV@addtomacro@n: protecting assignments from # .....	31
\XKV@addtomacro@o: protecting assignments from # .....	31
\XKV@ch@cksanitize: protecting assignments from # .....	34
\XKV@f@r: protecting assignments from # .....	30
\XKV@for@n: protecting assignments from # .....	30
\XKV@t@bulate: Solved bug occurring with empty prefix .....	61
<b>v2.6a</b>	<i>(2008/08/13)</i>
@s@lective@sanitize: Use private scratch register .....	32
\XKV@addtomacro@n: Use private scratch register .....	31
\XKV@addtomacro@o: Use private scratch register .....	31
\XKV@ch@cksanitize: Use private scratch register .....	34
\XKV@f@r: Use private scratch register .....	30
\XKV@for@n: Use private scratch register .....	30

## Index

Numbers written in *italics* refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in *roman* refer to the code lines where the entry is used.

<b>Symbols</b>	
\@fileswith@pti@ns .....	910, 1265
\@badrequireerror .....	866, 880
\@classoptionslist .....	842, 844, 847, 850, 1241, 1243
\@clsextension .....	688, 828, 1224, 1240
\@current .....	688, 859, 872, 881, 904, 1224
\@currname .....	859, 872, 881, 904
\@documentclasshook .....	1246
\@filelist .....	826

\@fileswith@pti@ns .....	866, 880, 910, 1239, 1265
\@firstofone .....	466, 511, 1015
\@firstoftwo .....	<u>33</u> , 41, 51, 66, 260, 261, 322, 464, 510, 528
\@ifncharacter .....	<u>59</u>
\@ifnextcharacter .....	<u>59</u> , 260, 261
\@onefilewithoptions ..	1245, 1249, 1256
\@onelevel@sanitize ..	140, 203, 777, 885, 916, 1027, 1056, 1117, 1226, 1234
\@pass@ptions .....	1204
\@pkgextension .....	1257
\@psset .....	<u>1295</u>
\@removeelement .....	918, 1235
\@s@l@ctive@sanitize ..	142, <u>147</u>
\@s@l@ective@sanitize ..	<u>135</u>
\@secondoftwo .....	.... <u>33</u> , 43, 49, 68, 320, 470, 487, 530
\@selective@sanitize ....	<u>135</u> , 195, 199
\@unknownoptionerror ..	689
\@unprocessedoptions ..	911
\@unusedoptionlist .....	.... 883, 885, 919, 1227, 1228, 1236
\@use@ption .....	1232
<b>C</b>	
columns (option) .....	24
\CurrentOption .....	638, 655, 656, 689, 694, 701, 890–893, 922, 1039, 1206, 1225, 1233, 1237, 1307
<b>D</b>	
\DeclareOptionX ..	17, <u>865</u> , 921, 1038, 1306
\DeclareOptionX* .....	18
default (option) .....	24
\define@boolkey .....	7, <u>396</u>
\define@boolkey+ .....	7
\define@boolkeys .....	8, <u>426</u> , 1051
\define@choicekey .....	5, <u>366</u>
\define@choicekey* .....	5
\define@choicekey*+ .....	6
\define@choicekey+ .....	6
\define@cmdkey .....	5, <u>351</u>
\define@cmdkeys .....	5, <u>358</u> , 1049
\define@key .....	4, <u>341</u> , 874
\delpresetkeys .....	15, <u>568</u>
\delsavekeys .....	12, <u>606</u>
\disable@keys .....	8, <u>533</u>
\documentclass .....	838
<b>E</b>	
\ExecuteOptionsX .....	18, <u>875</u>
<b>F</b>	
family (option) .....	24

file (option) .....	24
\filename@area .....	829, 832
\filename@base .....	829, 832
\filename@ext .....	828, 829, 832
\filename@parse .....	827
<b>G</b>	
\gdelpresetkeys .....	15, <u>568</u>
\gdelsavekeys .....	12, <u>606</u>
\global .....	13
\gpresetkeys .....	15, <u>552</u>
\gsavekeys .....	12, <u>594</u>
\gsavevalue .....	12, 330
\gunpresetkeys .....	16, <u>582</u>
\gunsavekeys .....	12, <u>616</u>
<b>I</b>	
\ifXKV@inpx .....	23, 686, 700
\ifXKV@knf .....	21, 523, 527, 679, 685
\ifXKV@pl ..	20, 241, 378, 390, 398, 416, 463, 469, 486, 496, 673, 1090, 1105
\ifXKV@preset .....	24, 672
\ifXKV@rkv .....	22, 714, 735, 789
\ifXKV@sg .....	19, 715, 790
\ifXKV@st .....	18, 243, 254, 353, 389, 415, 440, 446, 561, 588, 590, 599, 622, 693, 888, 1073, 1099
\ifXKV@colii 1043, 1134, 1139, 1178, 1187	
\ifXKV@coliii .....	.... 1044, 1135, 1139, 1178, 1188
\ifXKV@coliv 1045, 1135, 1140, 1179, 1189	
\ifXKV@colv 1046, 1136, 1141, 1179, 1190	
\ifXKV@view .....	1132, 1152, 1176
\ifXKV@vlabels .....	1180
\ifXKV@vwkey .....	1042, 1175
\ifXKV@wlabels .....	1191
\ignorespaces .....	1294
\immediate .....	1148, 1156, 1186
<b>K</b>	
\key@ifundefined .....	8, <u>512</u>
\KV@sp@b .....	936, 938, 940
\KV@sp@c .....	940, 943
\KV@sp@d .....	933, 934
\KV@sp@def .....	217, 263, 273, 296, 522, 661, 671, 932, 949, 959
\KV@def .....	972
\KV@default .....	957, 964
\KV@do .....	944, 947
\KV@err .....	<u>256</u>
\KV@errx .....	<u>256</u>
\KV@prefix .....	952, 966
\KV@split .....	946, 948

<b>O</b>	
\OptionNotUsed .....	1223
options:	
columns .....	24
default .....	24
family .....	24
file .....	24
prefix .....	24
type .....	24
view .....	25
vlabels .....	24
wcolsep .....	24
weol .....	24
wlabels .....	25
<b>P</b>	
\PackageError .....	825
\PackageWarning ...	824, 922, 1039, 1307
prefix (option) .....	24
\presetkeys .....	15, <u>552</u> , 1052
\ProcessOptionsX <i>18</i> , <u>876</u> , 924, 1041, 1309	
\ProcessOptionsX* .....	18
\pss@t .....	<u>1291</u>
\pss@t .....	27, <u>1291</u>
\pst@addfams .....	27, <u>1285</u>
\pst@famlist .....	
....	27, <u>1284</u> , 1287, 1288, 1292, 1296
\PSTXKeyLoaded .....	1269, 1305
<b>S</b>	
\savekeys .....	12, <u>594</u>
\savevalue .....	11, 329
\setkeys .....	9, <u>626</u> , 1116, 1294, 1296
\setkeys* .....	9
\setkeys*+ .....	10
\setkeys+ .....	10
\setrmkeys .....	10, <u>804</u>
\setrmkeys* .....	10
<b>T</b>	
type (option) .....	24
<b>U</b>	
\unpresetkeys .....	16, <u>582</u>
\unsavekeys .....	12, <u>616</u>
\usevalue .....	13, 752, 754
<b>V</b>	
view (option) .....	25
vlabels (option) .....	24
<b>W</b>	
wcolsep (option) .....	24
weol (option) .....	24
wlabels (option) .....	25
\write .....	1186
<b>X</b>	
\XKeyValLoaded .....	3, 815, 1277
\XKV@@@dx .....	870
\XKV@ch@ckch@ic@ .....	499, <u>510</u>
\XKV@ch@ckch@ice .....	497, <u>510</u>
\XKV@ch@ckchoice .....	453, <u>479</u>
\XKV@d@x .....	870
\XKV@ifcmd .....	<u>317</u>
\XKV@t@st@pte .....	<u>852</u>
\XKV@addtolist@n .....	<u>120</u> , 895
\XKV@addtolist@o .....	
.....	<u>127</u> , 236, 238, 241, 252, 694, 847, 903
\XKV@addtolist@x ...	<u>134</u> , 632, 766, 1057
\XKV@addtomacro@n .....	<u>113</u> ,
.....	118, 122, 124, 464, 466, 470, 474, 755
\XKV@addtomacro@o ...	<u>117</u> , 129, 131, 218
\XKV@afterelsefi .....	<u>35</u> ,
.....	37, 378, 398, 450, 719, 763, 794, 1090
\XKV@afterfi .....	<u>35</u> ,
.....	380, 400, 452, 726, 757, 765, 798, 1092
\XKV@cc .....	6, <u>437</u>
\XKV@cc* .....	6
\XKV@cc*+ .....	6
\XKV@cc+ .....	6
\XKV@ch@ckch@ice .....	442, 451, <u>456</u>
\XKV@ch@ckchoice .....	444, <u>448</u>
\XKV@ch@cksanitize ..	193, 194, 198, <u>201</u>
\XKV@checkchoice .....	
.....	391, 418, 422, 437, <u>438</u> , 1107, 1111
\XKV@checksantizea .	<u>192</u> , 226, 560, 628
\XKV@checksantizeb .....	
.....	<u>197</u> , 246, 302, 515, 535, 598
\XKV@classoptionslist .....	
.....	3, 840, 842, 883, 900
\XKV@d@f@n@ch@ic@k@y ...	385, 386, <u>387</u>
\XKV@d@f@ne@b@olkey .....	
.....	406, 410, <u>413</u> , 434, <u>1061</u>
\XKV@d@f@ne@boolkey .....	399, <u>409</u>
\XKV@d@f@ne@ch@ic@k@y ..	379, <u>386</u> , 1091
\XKV@d@f@ne@ch@ic@key ..	381, <u>385</u> , 1093
\XKV@d@fine@boolkey .....	401, <u>405</u>
\XKV@d@fine@boolkeys .....	430, <u>433</u>
\XKV@d@fine@ch@ic@key .....	
.....	371, 375, <u>377</u> , <u>1061</u>
\XKV@d@fine@ch@icekey ..	371, <u>373</u> , <u>1061</u>
\XKV@d@fine@choicekey .....	367, <u>368</u>
\XKV@d@fine@k@y .....	343, 347, <u>1061</u>
\XKV@d@fine@key .....	<u>347</u>
\XKV@d@l@presetkeys .....	571, 572, <u>574</u>
\XKV@d@x .....	867, <u>870</u>
\XKV@default .....	723, <u>773</u>
\XKV@define@boolkey .....	396, <u>397</u>

\XKV@define@boolkeys	426, <u>427</u>	\XKV@rm	25, 634, 694, 807, 908
\XKV@define@choicekey	366, <u>367</u>	\XKV@st@tk@ys	656, <u>659</u>
\XKV@define@cmdkey	351, <u>352</u> , 362, <u>1061</u>	\XKV@st@tk@ys@	672, 676, 681, <u>707</u>
\XKV@define@cmdkeys	358, <u>359</u>	\XKV@st@tkeys	636, 647, <u>653</u>
\XKV@define@default	337, 348, 353, 374, 415, 1069, 1075, 1085, 1101	\XKV@savekeys	594, 595, <u>596</u>
\XKV@define@key	341, <u>342</u> , 544, 546, <u>1061</u>	\XKV@setkeys	626, <u>627</u> , 806, 875, 905
\XKV@delete	245, 578, 612	\XKV@setrmkeys	804, <u>805</u>
\XKV@delpresetkeys	568, 569, <u>570</u>	\XKV@sp@deflist	214, 303, 360, 370, 428, 516, 654, 862
\XKV@delsavekeys	606, 607, <u>608</u>	\XKV@srstate	278, 727, 730, 795, 797, 799, 801
\XKV@depth	17, 279, 285, 634, 727, 730, 795, 797, 799, 801	\XKV@t@st@pte	852
\XKV@disable@keys	533, <u>534</u>	\XKV@t@stopta	287, 396
\XKV@documentclass	3, 700, 826, 831, 837, 839, 882	\XKV@t@stoptb	292
\XKV@doux	867, <u>869</u>	\XKV@t@stoptc	299
\XKV@doux	687, 691, 823, 869, 907	\XKV@t@stoptd	307
\XKV@err	256, 266, 473, 490, 538, 576, 586, 610, 620, 665, 696, 721, 759, 764, <u>824</u> , 838, 954, 968	\XKV@t@stopte	852, 875
\XKV@f@r	77, <u>81</u> , 92, 94	\XKV@tempa@toks	16, 73, 82, 114, 115, 143, 144, 202, 207, 212
\XKV@fams	302, 304, 515, 517, 523, 642, 674, 679, 696, 862	\XKV@testclass	700, 881, 882
\XKV@for@break	75, <u>90</u>	\XKV@testopta	287, 366, 437, 626, 804
\XKV@for@en	92	\XKV@testoptb	292, 308, 341, 366, 533, 552, 553, 568, 569, 582, 583, 594, 595, 606, 607, 616, 617
\XKV@for@eo	93, 642, 674	\XKV@testoptc	299, 626, 804
\XKV@for@en	72, 91, 216, 280, 655, 845, 1286	\XKV@testoptd	307, 351, 358, 396, 426
\XKV@for@o	91, 227, 232, 249, 361, 429, 536, 630, 736, 1149	\XKV@testopte	852, 876
\XKV@g@tkeyname	327, <u>328</u> , 631, 660, 892	\XKV@testsavekey	711, <u>734</u> , 786
\XKV@getkeyname	327, 565, 579	\XKV@unpresetkeys	582, 583, <u>584</u>
\XKV@getsg	334, 603, 613	\XKV@unsavekeys	616, 617, <u>618</u>
\XKV@ifcmd	317, 329, 330, 335, 737	\XKV@useoption	701, 894, <u>913</u>
\XKV@ifplus	260, 291	\XKV@usepresetkeys	635, 637, <u>640</u>
\XKV@ifstar	260, 288, 853, 867	\XKV@warn	256, 541, <u>824</u>
\XKV@ifundefined	37, 55, 525, 537, 543, 559, 575, 585, 597, 609, 619, 644, 708, 710, 720, 758, 785, 829, 884, 915	\XKV@wh@l@st	102, 107, <u>112</u>
\XKV@key@if@ndefined	518, <u>520</u> , 893	\XKV@wh@list	97, <u>99</u>
\XKV@key@ifundefined	512, <u>513</u>	\XKV@whilist	96, 523, 679, 826
\XKV@makehd	272, 295, 524, 643, 675, 680	\XKVV@columns	1118, 1123, 1125, 1127, 1129
\XKV@makepf	262, 294, 301, 514, 857	\XKVV@db	1048, 1057, 1149
\XKV@merge	225, 564, 602	\XKVV@default	1117, 1171, 1173
\XKV@na	654, 668	\XKVV@family	1163, 1165
\XKV@naa	629, 632, 648	\XKVV@file	1148, 1156, 1185
\XKV@pox	876, <u>877</u>	\XKVV@out	1047, 1148, 1156, 1186
\XKV@pr@setkeys	555, 556, <u>558</u>	\XKVV@prefix	1159, 1161
\XKV@prefix	263, 265, 267, 269, 275, 1057, 1058	\XKVV@t@bulate	1054
\XKV@presetkeys	552, 553, <u>554</u>	\XKVV@tabulate	1054, 1063, 1068, 1074, 1077, 1084, 1089, 1100, 1103
\XKV@r@placepointers	749	\XKVV@type	1167, 1169
\XKV@replacepointers	718, <u>749</u> , 793	\XKVV@wcolsep	1187–1190
		\XKVV@weol	1192
		\XKVV@xkvview	1150, <u>1158</u>
		\xkvview	24, <u>1115</u>