Documentation

# The xkeyval package *

Hendri Adriaens

v2.8 (2020/11/20)

**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



*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.



xkeyval

# 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 LaTeX $2_\varepsilon$ 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,[1] plain TeX users do `\input xkeyval`. LaTeX users do one of the following: `\usepackage{xkeyval}` or `\RequirePackage{xkeyval}`. The package does not have options. It is mandatory for 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 $\varepsilon$-TeX engine when available. In particular, `\ifcsname` is used whenever possible to avoid filling 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 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 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.

[1] 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 \⟨*prefix*⟩@⟨*family*⟩@⟨*key*⟩ with one argument holding ⟨*function*⟩. The default value for ⟨*prefix*⟩ is KV. This is the standard throughout the package to simplify mixing keyval and xkeyval keys. When ⟨*key*⟩ is used in a \setkeys command (see section 4) containing key=value, the macro \⟨*prefix*⟩@⟨*family*⟩@⟨*key*⟩ receives value as its argument and will be executed. The argument can be accessed by ⟨*function*⟩ 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 \⟨*prefix*⟩@⟨*family*⟩@⟨*key*⟩@default as a macro with no arguments and definition \⟨*prefix*⟩@⟨*family*⟩@⟨*key*⟩{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 ⟨*prefix*⟩ is specified and empty, the macros created by \define@key will have the form \⟨*family*⟩@⟨*key*⟩. When ⟨*family*⟩ is empty, the resulting form will be \⟨*prefix*⟩@⟨*key*⟩. When both ⟨*prefix*⟩ and ⟨*family*⟩ are empty, the form is \⟨*key*⟩. 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 ⟨*family*⟩ is to create distinct sets of keys. This can be used to avoid a macro setting keys meant for another macro only. The optional ⟨*prefix*⟩ 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.[2]

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.

[2] Like PSTricks, which uses a system originating from keyval, but which has been modified to use no families and psset 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*⟩@.[3] The two keys in the following example hence do exactly the same thing.[4]

```
\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*⟩

[3] Remember that some rules are applied when creating the header, the combination of ⟨*prefix*⟩ and ⟨*header*⟩. See section 3.1.

[4] Notice however, that the first key will be listed as a 'command key' by `xkvview` and the second as an 'ordinary key'. See section 12.1.

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

The optional ⟨*bin*⟩ 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 ⟨*al*⟩ 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 `\ifcase` statement in ⟨*func*⟩.

```
\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*⟩}

`\define@choicekey+`
`\define@choicekey*+`

These macros operate as their counterparts without the +, but allow for specifying two key macros. ⟨*f1*⟩ will be executed when the input was correct and ⟨*f2*⟩ will be executed when the input was not allowed. Again, the starred version executes the check after converting user input and ⟨*al*⟩ 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*⟩}

`\XKV@cc`
`\XKV@cc*`
`\XKV@cc+`
`\XKV@cc*+`

Choice keys work by adding (an internal version[5] of) the `\XKV@cc` macro to key macros. This macro has similar arguments as the `\define@choicekey` macro and

[5] 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*⟩[3,6] if ⟨*mp*⟩ is not specified, using `\newif`[7] (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}%
}
```

[6] 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.

[7] The LaTeX of 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 \⟨*prefix*⟩@⟨*family*⟩@⟨*key*⟩[3] where ⟨*family*⟩ is an element of the list ⟨*families*⟩ and `key` is an element of the options list ⟨*keys*⟩ and not of ⟨*na*⟩. 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 ⟨*families*⟩, the first family from the list defining the key will set it. No errors are produced when ⟨*keys*⟩ is empty. If ⟨*family*⟩ is empty, the macro will set keys of the form \⟨*prefix*⟩@⟨*key*⟩. If ⟨*prefix*⟩ is specified and empty, the macro will set keys of the form \⟨*family*⟩@⟨*key*⟩. If both ⟨*prefix*⟩ and ⟨*family*⟩ are empty, the macro will set keys of the form \⟨*key*⟩.

```
\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 `\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 `\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 ⟨*na*⟩ will be ignored fully and will not be appended to the `\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 `\XKV@rm` (so `\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.[8] 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.

[8] 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` `\gsavekeys` 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). 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` `\gdelsavekeys` `\unsavekeys` `\gunsavekeys` 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 ⟨*family*⟩ (or ⟨*families*⟩) 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 ⟨*family*⟩ (or ⟨*families*⟩) 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 ⟨*prefix*⟩. 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 `[`⟨*arg*⟩`]` and `<`⟨*arg*⟩`>` 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[⟨prefix⟩]<⟨family⟩>{⟨key⟩}[⟨default⟩]{⟨function⟩}
```

`\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 ⟨*function*⟩. This will contain ⟨*default*⟩ when no value has been specified for the key. The value of the optional argument ⟨*default*⟩ is empty by default. This implies that when the user does not assign a value to ⟨*key*⟩ and when no default value has been defined, no error will be produced. The optional argument ⟨*family*⟩ 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.[9]

```
% 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 14.7 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

[9] 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.[10] 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}
```

[10] 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 LaTeX option macros (see section 7) are not protected as LaTeX 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 pstricks-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,[11] 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*
`prefix`
`family`
`type`
`default`

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]`.

```
\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*
`columns`

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).

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

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

*option*
`vlabels`

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

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

*options*
`file`
`wcolsep`
`weol`

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

[11] 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 ⟨*prefix*⟩-⟨*family*⟩-⟨*keyname*⟩ 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 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.[12] 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

[12] The xkvltxp package consists of the file `xkvltxp.sty`.

to do \RequirePackage{xkvltxp} on the first line of the LaTeX 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[13] 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}
```

[13] 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.

```
%<*xkvtex>
\csname XKeyValLoaded\endcsname
\let\XKeyValLoaded\endinput
```

Adjust some catcodes to define internal macros.

```
\edef\XKVcatcodes{%
  \catcode‘\noexpand\@\the\catcode‘\@\relax
  \catcode‘\noexpand\=\the\catcode‘\=\relax
  \catcode‘\noexpand\,\the\catcode‘\,\relax
  \catcode‘\noexpand\:\the\catcode‘\:\relax
  \let\noexpand\XKVcatcodes\relax
}
\catcode‘\@11\relax
\catcode‘\=12\relax
\catcode‘\,12\relax
\catcode‘\:12\relax
```

Load some basic utilities.

```
\input xkvutils
```

Initializations.

```
\newcount\XKV@depth
\newif\ifXKV@st
\newif\ifXKV@sg
\newif\ifXKV@pl
\newif\ifXKV@knf
\newif\ifXKV@rkv
\newif\ifXKV@inpox
\newif\ifXKV@preset
\let\XKV@rm\@empty
```

Load LaTeX primitives if necessary and provide information.

```
\ifx\ProvidesFile\@undefined
  \message{2014/12/03 v2.7a key=value parser (HA)}
  \input xkvtxhdr
\else
  \ProvidesFile{xkeyval.tex}[2014/12/03 v2.7a key=value parser (HA)]
  \@addtofilelist{xkeyval.tex}
\fi
```

\XKV@warn
\XKV@err
\KV@err
\KV@errx

Warning and error macros. We redefine the keyval error macros to use the xkeyval ones. This avoids redefining them again when we redefine the \XKV@warn and \XKV@err macros in xkeyval.sty.

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

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

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

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

```
\def\XKV@makepf#1{%
  \KV@@sp@def\XKV@prefix{#1}%
  \def\XKV@resa{XKV}%
  \ifx\XKV@prefix\XKV@resa
    \XKV@err{'XKV' prefix is not allowed}%
    \let\XKV@prefix\@empty
  \else
    \edef\XKV@prefix{\ifx\XKV@prefix\@empty\else\XKV@prefix @\fi}%
  \fi
}
```

`\XKV@makehd` `{`⟨*family*⟩`}`

Creates the header, like `prefix@family@` in `\prefix@family@key`. If ⟨*family*⟩ is empty, the header reduces to `prefix@`.

```
\def\XKV@makehd#1{%
  \expandafter\KV@@sp@def\expandafter\XKV@header\expandafter{#1}%
  \edef\XKV@header{%
    \XKV@prefix\ifx\XKV@header\@empty\else\XKV@header @\fi
  }%
}
```

`\XKV@srstate` `{`⟨*postfix1*⟩`}{`⟨*postfix2*⟩`}`

Macro to save and restore xkeyval internals to allow for nesting `\setkeys` commands. 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.

```
\def\XKV@srstate#1#2{%
  \ifx\@empty#2\@empty\advance\XKV@depth\@ne\fi
  \XKV@for@n{XKV@prefix,XKV@fams,XKV@tkey,XKV@na,%
    ifXKV@st,ifXKV@pl,ifXKV@knf,CurrentOption}\XKV@resa{%
    \expandafter\let\csname\XKV@resa#1\expandafter
      \endcsname\csname\XKV@resa#2\endcsname
  }%
  \ifx\@empty#1\@empty\advance\XKV@depth\m@ne\fi
}
```

`\XKV@testopta` `{`⟨*function*⟩`}`
`\XKV@t@stopta` Tests for the presence of an optional star or plus and executes ⟨*function*⟩ afterwards.

```
\def\XKV@testopta#1{%
  \XKV@ifstar{\XKV@sttrue\XKV@t@stopta{#1}}%
    {\XKV@stfalse\XKV@t@stopta{#1}}%
}
\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*⟩.

```
\def\XKV@testoptb#1{\@testopt{\XKV@t@stoptb#1}{KV}}
\def\XKV@t@stoptb#1[#2]#3{%
```

Set prefix.

```
  \XKV@makepf{#2}%
```

Set header.

```
  \XKV@makehd{#3}%
```

Save family name for later use.

```
  \KV@@sp@def\XKV@tfam{#3}%
  #1%
}
```

`\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*⟩.

```
\def\XKV@testoptc#1{\@testopt{\XKV@t@stoptc#1}{KV}}
\def\XKV@t@stoptc#1[#2]#3{%
  \XKV@makepf{#2}%
  \XKV@checksanitizeb{#3}\XKV@fams
  \expandafter\XKV@sp@deflist\expandafter
    \XKV@fams\expandafter{\XKV@fams}%
  \@testopt#1{}%
}
```

`\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.

```
\def\XKV@testoptd#1#2{%
  \XKV@testoptb{%
    \edef\XKV@tempa{#2\XKV@header}%
    \def\XKV@tempb{\@testopt{\XKV@t@stoptd#1}}%
    \expandafter\XKV@tempb\expandafter{\XKV@tempa}%
  }%
}
\def\XKV@t@stoptd#1[#2]#3{%
```

`\ifXKV@st` gives the presence of an optional default value.

```
  \@ifnextchar[{\XKV@sttrue#1{#2}{#3}}{\XKV@stfalse#1{#2}{#3}[]}%
}
```

`\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. Otherwise, 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.

```
\def\XKV@ifcmd#1#2#3{%
  \def\XKV@@ifcmd##1#2##2##3\@nil##4{%
    \def##4{##2}\ifx##4\@nnil
      \def##4{##1}\expandafter\@secondoftwo
    \else
      \expandafter\@firstoftwo
    \fi
  }%
  \XKV@@ifcmd#1#2{\@nil}\@nil#3%
}
```

`\XKV@getkeyname` ⟨*keyvalue*⟩⟨*bin*⟩

Utility macro to retrieve the key name from ⟨*keyvalue*⟩ 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 ⟨*bin*⟩.

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

`\XKV@g@tkeyname` ⟨*key*⟩=⟨*value*⟩`\@nil`⟨*bin*⟩

Use `\XKV@ifcmd` several times to check the syntax of ⟨*value*⟩. Save ⟨*key*⟩ to ⟨*bin*⟩.

```
\long\def\XKV@g@tkeyname#1=#2\@nil#3{%
  \XKV@ifcmd{#1}\savevalue#3{\XKV@rkvtrue\XKV@sgfalse}{%
    \XKV@ifcmd{#1}\gsavevalue#3%
      {\XKV@rkvtrue\XKV@sgtrue}{\XKV@rkvfalse\XKV@sgfalse}%
  }%
}
```

`\XKV@getsg` ⟨*key*⟩⟨*bin*⟩

Utility macro to check whether `key` or `\global{key}` has been specified in ⟨*key*⟩. The key name is saved to ⟨*bin*⟩

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

`\XKV@define@default` {⟨*key*⟩}{⟨*default*⟩}

Defines the default value macro for ⟨*key*⟩ and given `\XKV@header`.

```
\def\XKV@define@default#1#2{%
  \expandafter\def\csname\XKV@header#1@default\expandafter
    \endcsname\expandafter{\csname\XKV@header#1\endcsname{#2}}%
}
```

`\define@key` [⟨*prefix*⟩]{⟨*family*⟩}

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 ⟨*prefix*⟩ and the mandatory ⟨*family*⟩.

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

`\XKV@define@key` `{`⟨*key*⟩`}`

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

```
\def\XKV@define@key#1{%
  \@ifnextchar[{\XKV@d@fine@k@y{#1}}{%
    \long\expandafter\def\csname\XKV@header#1\endcsname####1%
  }%
}
```

`\XKV@d@fine@key` `{`⟨*key*⟩`}[`⟨*default*⟩`]`

Defines the key macro and the default value macro.

```
\def\XKV@d@fine@k@y#1[#2]{%
  \XKV@define@default{#1}{#2}%
  \expandafter\def\csname\XKV@header#1\endcsname##1%
}
```

`\define@cmdkey` `[`⟨*prefix*⟩`]{`⟨*family*⟩`}[`⟨*mp*⟩`]{`⟨*key*⟩`}`

Define a command key. Test for optional ⟨*prefix*⟩, mandatory ⟨*family*⟩, optional ⟨*mp*⟩ 'macro prefix' and mandatory ⟨*key*⟩ name.

```
\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*⟩.

```
\def\XKV@define@cmdkey#1#2[#3]#4{%
  \ifXKV@st\XKV@define@default{#2}{#3}\fi
  \def\XKV@tempa{\expandafter\def\csname\XKV@header#2\endcsname####1}%
  \begingroup\expandafter\endgroup\expandafter\XKV@tempa\expandafter
    {\expandafter\def\csname#1#2\endcsname{##1}#4}%
}
```

`\define@cmdkeys` `[`⟨*prefix*⟩`]{`⟨*family*⟩`}[`⟨*mp*⟩`]{`⟨*keys*⟩`}`

Define multiple command keys.

```
\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.

```
\def\XKV@define@cmdkeys#1#2[#3]{%
  \XKV@sp@deflist\XKV@tempa{#2}%
  \XKV@for@o\XKV@tempa\XKV@tempa{%
    \edef\XKV@tempa{\noexpand\XKV@define@cmdkey{#1}{\XKV@tempa}}%
    \XKV@tempa[#3]{}%
  }%
}
```

`\define@choicekey` `*+[`⟨*prefix*⟩`]{`⟨*family*⟩`}`

Choice keys. First check optional star, plus and prefix and store the family.

```
\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.

```
\def\XKV@define@choicekey#1{\@testopt{\XKV@d@fine@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.

```
\def\XKV@d@fine@choicekey#1[#2]#3{%
  \toks@{#2}%
  \XKV@sp@deflist\XKV@tempa{#3}\XKV@toks\expandafter{\XKV@tempa}%
  \@ifnextchar[{\XKV@d@fine@ch@icekey{#1}}{\XKV@d@fine@ch@ic@key{#1}}%
}
```

`\XKV@d@fine@ch@icekey` `{⟨key⟩}[⟨default⟩]`

Define the default value macro if a default value was specified.

```
\def\XKV@d@fine@ch@icekey#1[#2]{%
  \XKV@define@default{#1}{#2}%
  \XKV@d@fine@ch@ic@key{#1}%
}
```

`\XKV@d@fine@ch@ic@key` `{⟨key⟩}`

Eat correct number of arguments.

```
\def\XKV@d@fine@ch@ic@key#1{%
  \ifXKV@pl\XKV@afterelsefi
    \expandafter\XKV@d@f@ne@ch@ic@k@y
  \else\XKV@afterfi
    \expandafter\XKV@d@f@ne@ch@ic@key
  \fi
  \csname\XKV@header#1\endcsname
}
```

`\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.

```
\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. ⟨*fucntion1*⟩ will be executed on correct input, ⟨*function2*⟩ on incorrect input.

```
\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 execute one of its mandatory arguments.

```
\def\XKV@d@f@n@@ch@ic@k@y#1#2{%
  \edef#1##1{%
    \ifXKV@st\noexpand\XKV@sttrue\else\noexpand\XKV@stfalse\fi
    \ifXKV@pl\noexpand\XKV@pltrue\else\noexpand\XKV@plfalse\fi
    \noexpand\XKV@checkchoice[\the\toks@]{##1}{\the\XKV@toks}%
  }%
  \def\XKV@tempa{\def#1####1}%
  \expandafter\XKV@tempa\expandafter{#1{##1}#2}%
}
```

`\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.

```
\def\define@boolkey{\XKV@t@stopta{\XKV@testoptd\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.

```
\def\XKV@define@boolkey#1#2[#3]{%
  \ifXKV@pl\XKV@afterelsefi
    \expandafter\XKV@d@f@ne@boolkey
  \else\XKV@afterfi
    \expandafter\XKV@d@fine@boolkey
  \fi
  \csname\XKV@header#2\endcsname{#2}{#1#2}{#3}%
}
```

`\XKV@d@fine@boolkey` ⟨*key macro*⟩{⟨*key*⟩}{⟨*if name*⟩}{⟨*default*⟩}{⟨*function*⟩}

Eat one mandatory key function and pass it. Insert 'setting the if'.

```
\def\XKV@d@fine@boolkey#1#2#3#4#5{%
  \XKV@d@f@ne@b@olkey#1{#2}{#3}{#4}%
    {{\csname#3\XKV@resa\endcsname#5}}%
}
```

`\XKV@d@f@ne@boolkey` ⟨*key macro*⟩{⟨*key*⟩}{⟨*if name*⟩}{⟨*default*⟩}{⟨*func1*⟩}{⟨*func2*⟩}

Eat two mandatory key functions and pass them. Insert 'setting the if'.

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

`\XKV@d@f@ne@b@olkey` ⟨*key macro*⟩{⟨*key*⟩}{⟨*if name*⟩}{⟨*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.

```
\def\XKV@d@f@ne@b@olkey#1#2#3#4#5{%
  \expandafter\newif\csname if#3\endcsname
  \ifXKV@st\XKV@define@default{#2}{#4}\fi
  \ifXKV@pl
    \def#1##1{\XKV@pltrue\XKV@sttrue
      \XKV@checkchoice[\XKV@resa]{##1}{true,false}#5%
    }%
  \else
    \def#1##1{\XKV@plfalse\XKV@sttrue
      \XKV@checkchoice[\XKV@resa]{##1}{true,false}#5%
    }%
  \fi
}
```

`\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.

```
\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.

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

```
  \XKV@sp@deflist\XKV@tempa{#2}%
  \XKV@for@o\XKV@tempa\XKV@tempa{%
    \expandafter\XKV@d@fine@boolkeys\expandafter{\XKV@tempa}{#1}{#3}%
  }%
}
```

`\XKV@d@fine@boolkeys` `{`⟨*key*⟩`}{`⟨*mp*⟩`}{`⟨*default*⟩`}`

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

```
\def\XKV@d@fine@boolkeys#1#2#3{%
  \expandafter\XKV@d@f@ne@b@olkey\csname\XKV@header#1\endcsname
    {#1}{#2#1}{#3}{{\csname#2#1\XKV@resa\endcsname}}%
}
```

`\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 `+`.

```
\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`.

```
\def\XKV@checkchoice[#1]#2#3{%
  \def\XKV@tempa{#1}%
  \ifXKV@st\lowercase{\fi
  \ifx\XKV@tempa\@empty
    \def\XKV@tempa{\XKV@ch@ckch@ice\@nil{#2}{#3}}%
  \else
    \def\XKV@tempa{\XKV@ch@ckchoice#1\@nil{#2}{#3}}%
  \fi
  \ifXKV@st}\fi\XKV@tempa
}
```

`\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.

```
\def\XKV@ch@ckchoice#1#2\@nil#3#4{%
  \def\XKV@tempa{#2}%
  \ifx\XKV@tempa\@empty\XKV@afterelsefi
    \XKV@ch@ckch@ice#1{#3}{#4}%
  \else\XKV@afterfi
    \XKV@@ch@ckchoice#1#2{#3}{#4}%
  \fi
}
```

`\XKV@ch@ckch@ice` ⟨*bin*⟩`{`⟨*input*⟩`}{`⟨*allowed*⟩`}`

Checks whether ⟨*input*⟩ is in the list ⟨*allowed*⟩ and perform actions accordingly.

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

If we have a ⟨*bin*⟩, store the input there.

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

The ⟨*input*⟩ is allowed.

```
  \ifXKV@pl
```

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

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

Else, we have one function; execute it.

```
    \XKV@addtomacro@n\XKV@tempa\@firstofone
  \fi
\else
```

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

```
  \ifXKV@pl
    \XKV@addtomacro@n\XKV@tempa\@secondoftwo
  \else
```

Else, raise an error and gobble the one function.

```
    \XKV@toks{#2}%
    \XKV@err{value `\the\XKV@toks' is not allowed}%
    \XKV@addtomacro@n\XKV@tempa\@gobble
  \fi
\fi
\XKV@tempa
}
```

`\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.

```
\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).

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

The input.

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

Define the while loop.

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

The ⟨*input*⟩ was not in ⟨*allowed*⟩. Set the number to -1.

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

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

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

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

```
        \XKV@toks{#3}%
        \XKV@err{value `\the\XKV@toks' is not allowed}%
        \let\XKV@tempd\@gobble
      \fi
    \else
      \ifx#1\XKV@tempb
```

We found ⟨*input*⟩ in ⟨*allowed*⟩. Save the number of the ⟨*input*⟩ in the list ⟨*allowed*⟩.

```
        \edef#2{\the\count@}\count@\XKV@tempa
        \ifXKV@pl
          \let\XKV@tempd\XKV@@ch@ckch@ice
        \else
          \let\XKV@tempd\XKV@@ch@ckch@ic@
        \fi
      \else
```

Increase counter and check next item in the list ⟨*allowed*⟩.

```
        \advance\count@\@ne
        \let\XKV@tempd\XKV@tempc
      \fi
    \fi
    \XKV@tempd
  }%
```

Start the while loop.

```
  \XKV@tempc#4,\@nil,%
}
```

`\XKV@@ch@ckch@ice` `\XKV@@ch@ckch@ic@`

⟨*text*⟩`\@nil,`

Gobble remaining ⟨*text*⟩ and execute the proper key function.

```
\def\XKV@@ch@ckch@ice#1\@nil,{\@firstoftwo}
\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.

```
\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.

```
\def\XKV@key@ifundefined[#1]#2{%
  \XKV@makepf{#1}%
  \XKV@checksanitizeb{#2}\XKV@fams
  \expandafter\XKV@sp@deflist\expandafter
    \XKV@fams\expandafter{\XKV@fams}%
  \XKV@key@if@ndefined
}
```

`\XKV@key@if@ndefined`

`{`⟨*key*⟩`}`

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

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

Loop over possible families.

```
  \XKV@whilist\XKV@fams\XKV@tfam\ifXKV@knf\fi{%
```

Set the header.

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

Check whether the macro for the key is defined.

```
    \XKV@ifundefined{\XKV@header\XKV@tkey}{}{\XKV@knffalse}%
  }%
```

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

```
  \ifXKV@knf
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
}
```

`\disable@keys` `[`⟨*prefix*⟩`]{`⟨*family*⟩`}`

Macro that make a key produce a warning on use.

```
\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.

```
\def\XKV@disable@keys#1{%
  \XKV@checksanitizeb{#1}\XKV@tempa
  \XKV@for@o\XKV@tempa\XKV@tempa{%
    \XKV@ifundefined{\XKV@header\XKV@tempa}{%
      \XKV@err{key ‘\XKV@tempa’ undefined}%
    }{%
      \edef\XKV@tempb{%
        \noexpand\XKV@warn{key ‘\XKV@tempa’ has been disabled}%
      }%
      \XKV@ifundefined{\XKV@header\XKV@tempa @default}{%
        \edef\XKV@tempc{\noexpand\XKV@define@key{\XKV@tempa}}%
      }{%
        \edef\XKV@tempc{\noexpand\XKV@define@key{\XKV@tempa}[]}%
      }%
      \expandafter\XKV@tempc\expandafter{\XKV@tempb}%
    }%
  }%
}
```

`\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.

```
\def\presetkeys{\XKV@stfalse\XKV@testoptb\XKV@presetkeys}
\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.

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

```
  \XKV@pr@setkeys{#1}{preseth}%
  \XKV@pr@setkeys{#2}{presett}%
}
```

`\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.

```
\def\XKV@pr@setkeys#1#2{%
  \XKV@ifundefined{XKV@\XKV@header#2}{%
    \XKV@checksanitizea{#1}\XKV@tempa
    \ifXKV@st\expandafter\global\fi\expandafter\def\csname
      XKV@\XKV@header#2\expandafter\endcsname\expandafter{\XKV@tempa}%
  }{%
    \expandafter\XKV@merge\csname XKV@\XKV@header
      #2\endcsname{#1}\XKV@getkeyname
  }%
}
```

`\delpresetkeys` `[`⟨*prefix*⟩`]{`⟨*family*⟩`}`
`\gdelpresetkeys`

Macros to remove entries from presets.

```
\def\delpresetkeys{\XKV@stfalse\XKV@testoptb\XKV@delpresetkeys}
\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.

```
\def\XKV@delpresetkeys#1#2{%
  \XKV@d@lpresetkeys{#1}{preseth}%
  \XKV@d@lpresetkeys{#2}{presett}%
}
```

`\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.

```
\def\XKV@d@lpresetkeys#1#2{%
  \XKV@ifundefined{XKV@\XKV@header#2}{%
    \XKV@err{no presets defined for `\XKV@header'}%
  }{%
    \expandafter\XKV@delete\csname XKV@\XKV@header
      #2\endcsname{#1}\XKV@getkeyname
  }%
}
```

`\unpresetkeys` `[`⟨*prefix*⟩`]{`⟨*family*⟩`}`
`\gunpresetkeys`

Removes presets for a particular family.

```
\def\unpresetkeys{\XKV@stfalse\XKV@testoptb\XKV@unpresetkeys}
\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`).

```
\def\XKV@unpresetkeys{%
  \XKV@ifundefined{XKV@\XKV@header preseth}{%
    \XKV@err{no presets defined for `\XKV@header'}%
  }{%
    \ifXKV@st\expandafter\global\fi\expandafter\let
      \csname XKV@\XKV@header preseth\endcsname\@undefined
    \ifXKV@st\expandafter\global\fi\expandafter\let
      \csname XKV@\XKV@header presett\endcsname\@undefined
  }%
}
```

`\savekeys` `\gsavekeys` `[⟨prefix⟩]{⟨family⟩}`

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

```
\def\savekeys{\XKV@stfalse\XKV@testoptb\XKV@savekeys}
\def\gsavekeys{\XKV@sttrue\XKV@testoptb\XKV@savekeys}
```

`\XKV@savekeys` `{⟨key list⟩}`

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

```
\def\XKV@savekeys#1{%
  \XKV@ifundefined{XKV@\XKV@header save}{%
    \XKV@checksanitizeb{#1}\XKV@tempa
    \ifXKV@st\expandafter\global\fi\expandafter\def\csname XKV@%
      \XKV@header save\expandafter\endcsname\expandafter{\XKV@tempa}%
  }{%
    \expandafter\XKV@merge\csname XKV@\XKV@header
      save\endcsname{#1}\XKV@getsg
  }%
}
```

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

Remove entries from the list of save keys.

```
\def\delsavekeys{\XKV@stfalse\XKV@testoptb\XKV@delsavekeys}
\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.

```
\def\XKV@delsavekeys#1{%
  \XKV@ifundefined{XKV@\XKV@header save}{%
    \XKV@err{no save keys defined for `\XKV@header'}%
  }{%
    \expandafter\XKV@delete\csname XKV@\XKV@header
      save\endcsname{#1}\XKV@getsg
  }%
}
```

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

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

```
\def\unsavekeys{\XKV@stfalse\XKV@testoptb\XKV@unsavekeys}
\def\gunsavekeys{\XKV@sttrue\XKV@testoptb\XKV@unsavekeys}
```

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

```
\def\XKV@unsavekeys{%
  \XKV@ifundefined{XKV@\XKV@header save}{%
    \XKV@err{no save keys defined for `\XKV@header'}%
  }{%
    \ifXKV@st\expandafter\global\fi\expandafter\let
      \csname XKV@\XKV@header save\endcsname\@undefined
  }%
}
```

`\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.

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

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

```
\long\def\XKV@setkeys[#1]#2{%
  \XKV@checksanitizea{#2}\XKV@resb
  \let\XKV@naa\@empty
```

Retrieve a list of key names from the user input.

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

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

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

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

```
  \XKV@usepresetkeys{#1}{preseth}%
  \expandafter\XKV@s@tkeys\expandafter{\XKV@resb}{#1}%
  \XKV@usepresetkeys{#1}{presett}%
  \let\CurrentOption\@empty
}
```

`\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.

```
\def\XKV@usepresetkeys#1#2{%
  \XKV@presettrue
  \XKV@for@eo\XKV@fams\XKV@tfam{%
    \XKV@makehd\XKV@tfam
    \XKV@ifundefined{XKV@\XKV@header#2}{}{%
      \XKV@toks\expandafter\expandafter\expandafter
        {\csname XKV@\XKV@header#2\endcsname}%
      \@expandtwoargs\XKV@s@tkeys{\the\XKV@toks}%
        {\XKV@naa\ifx\XKV@naa\@empty\else,\fi#1}%
    }%
  }%
  \XKV@presetfalse
}
```

`\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*⟩.

```
\long\def\XKV@s@tkeys#1#2{%
```

Define the list of key names which should be ignored.

```
  \XKV@sp@deflist\XKV@na{#2}%
```

Loop over the key=value list.

```
  \XKV@for@n{#1}\CurrentOption{%
```

Split key and value.

```
    \expandafter\XKV@s@tk@ys\CurrentOption==\@nil
  }%
}
```

`\XKV@s@tk@ys` ⟨*key*⟩=⟨*value*⟩=#3`\@nil`

Split key name and value (if present). If #3 non-empty, there was no =⟨*value*⟩.

```
\long\def\XKV@s@tk@ys#1=#2=#3\@nil{%
```

Check for `\savevalue` and `\gsavevalue` and remove spaces from around the key name.

```
  \XKV@g@tkeyname#1=\@nil\XKV@tkey
  \expandafter\KV@@sp@def\expandafter\XKV@tkey\expandafter{\XKV@tkey}%
```

If the key is empty and a value has been specified, generate an error.

```
  \ifx\XKV@tkey\@empty
    \XKV@toks{#2}%
    \ifcat$\the\XKV@toks$\else
      \XKV@err{no key specified for value `\the\XKV@toks'}%
    \fi
  \else
```

If in the `\XKV@na` list, ignore the key.

```
    \@expandtwoargs\in@{,\XKV@tkey,}{,\XKV@na,}%
    \ifin@\else
      \XKV@knftrue
      \KV@@sp@def\XKV@tempa{#2}%
      \ifXKV@preset\XKV@s@tk@ys@{#3}\else
        \ifXKV@pl
```

If a command with a + is used, set keys in all families on the list.

```
          \XKV@for@eo\XKV@fams\XKV@tfam{%
            \XKV@makehd\XKV@tfam
            \XKV@s@tk@ys@{#3}%
          }%
        \else
```

Else, scan the families on the list but stop when the key is found or when the list has run out.

```
          \XKV@whilist\XKV@fams\XKV@tfam\ifXKV@knf\fi{%
            \XKV@makehd\XKV@tfam
            \XKV@s@tk@ys@{#3}%
          }%
        \fi
      \fi
      \ifXKV@knf
        \ifXKV@inpox
```

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

```
463          \ifx\XKV@doxs\relax
```

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

```
464            \ifx\@currext\@clsextension\else
465              \let\CurrentOption\XKV@tkey\@unknownoptionerror
466            \fi
```

Pass the option through \DeclareOptionX*.

```
467          \else\XKV@doxs\fi
468        \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.

```
469          \ifXKV@st
470            \XKV@addtolist@o\XKV@rm\CurrentOption
471          \else
472            \XKV@err{‘\XKV@tkey’ undefined in families ‘\XKV@fams’}%
473          \fi
474        \fi
475      \else
```

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

```
476        \ifXKV@inpox\ifx\XKV@testclass\XKV@documentclass
477          \expandafter\XKV@useoption\expandafter{\CurrentOption}%
478        \fi\fi
479      \fi
480    \fi
481  \fi
482}
```

\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.

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

Check whether the key macro exists.

```
484  \XKV@ifundefined{\XKV@header\XKV@tkey}{}{%
485    \XKV@knffalse
```

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

```
486    \XKV@ifundefined{XKV@\XKV@header save}{}{%
487      \expandafter\XKV@testsavekey\csname XKV@\XKV@header
488        save\endcsname\XKV@tkey
489    }%
```

Save the value of a key.

```
490    \ifXKV@rkv
491      \ifXKV@sg\expandafter\global\fi\expandafter\let
492        \csname XKV@\XKV@header\XKV@tkey @value\endcsname\XKV@tempa
493    \fi
```

Replace pointers by saved values.

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

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

```
    \ifx\@empty#1\@empty\XKV@afterelsefi
      \XKV@ifundefined{\XKV@header\XKV@tkey @default}{%
        \XKV@err{no value specified for key `\XKV@tkey'}%
      }{%
        \expandafter\expandafter\expandafter\XKV@default
          \csname\XKV@header\XKV@tkey @default\endcsname\@nil
      }%
    \else\XKV@afterfi
```

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

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

Execute the key.

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

Restore the current state.

```
      \XKV@srstate{}{@\romannumeral\XKV@depth}%
    \fi
  }%
}
```

`\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.

```
\def\XKV@testsavekey#1#2{%
  \ifXKV@rkv\else
    \XKV@for@o#1\XKV@resa{%
      \expandafter\XKV@ifcmd\expandafter{\XKV@resa}\global\XKV@resa{%
        \ifx#2\XKV@resa
          \XKV@rkvtrue\XKV@sgtrue
        \fi
      }{%
        \ifx#2\XKV@resa
          \XKV@rkvtrue\XKV@sgfalse
        \fi
      }%
    }%
  \fi
}
```

`\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.

```
\long\def\XKV@replacepointers#1{%
  \let\XKV@tempa\@empty
  \let\XKV@resa\@empty
  \XKV@r@placepointers#1\usevalue\@nil
```

```
}
\long\def\XKV@r@placepointers#1\usevalue#2{%
  \XKV@addtomacro@n\XKV@tempa{#1}%
  \def\XKV@tempb{#2}%
  \ifx\XKV@tempb\@nnil\else\XKV@afterfi
    \XKV@ifundefined{XKV@\XKV@header#2@value}{%
      \XKV@err{no value recorded for key `#2'; ignored}%
      \XKV@r@placepointers
    }{%
      \@expandtwoargs\in@{,#2,}{,\XKV@resa,}%
      \ifin@\XKV@afterelsefi
        \XKV@err{back linking pointers; pointer replacement canceled}%
      \else\XKV@afterfi
        \XKV@addtolist@x\XKV@resa{#2}%
        \expandafter\expandafter\expandafter\XKV@r@placepointers
          \csname XKV@\XKV@header#2@value\endcsname
      \fi
    }%
  \fi
}
```

`\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.

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

Retrieve the first token in the macro.

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

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

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

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

```
  \@onelevel@sanitize\XKV@tempb
  \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.

```
    \begingroup
      \expandafter\def\csname\XKV@header\XKV@tkey\endcsname##1{%
        \gdef\XKV@tempa{##1}%
      }%
      \csname\XKV@header\XKV@tkey @default\endcsname
    \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.

```
    \XKV@ifundefined{XKV@\XKV@header save}{}{%
      \expandafter\XKV@testsavekey\csname XKV@\XKV@header
        save\endcsname\XKV@tkey
    }%
    \ifXKV@rkv
      \ifXKV@sg\expandafter\global\fi\expandafter\let
        \csname XKV@\XKV@header\XKV@tkey @value\endcsname\XKV@tempa
    \fi
```

Replace the pointers.

```
    \expandafter\XKV@replacepointers\expandafter
      {\XKV@tempa}\XKV@afterelsefi
```

Save internal state.

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

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

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

Restore internal state.

```
    \XKV@srstate{}{@\romannumeral\XKV@depth}%
  \else\XKV@afterfi
```

Save internal state.

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

Execute the key with the default value.

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

Restore the state.

```
    \XKV@srstate{}{@\romannumeral\XKV@depth}%
  \fi
}
```

`\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.

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

`\XKV@setrmkeys` `[`⟨*na*⟩`]`

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

```
\def\XKV@setrmkeys[#1]{%
  \def\XKV@tempa{\XKV@setkeys[#1]}%
  \expandafter\XKV@tempa\expandafter{\XKV@rm}%
}
```

Reset catcodes.

```
\XKVcatcodes
⟨/xkvtex⟩
```

## 14.2 xkeyval.sty

Initialize the package.

```
%<*xkvlatex>
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesPackage{xkeyval}
  [2020/11/20 v2.8 package option processing (HA)]
```

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

```
\ifx\XKeyValLoaded\endinput\else\input xkeyval \fi
\edef\XKVcatcodes{%
  \catcode`\noexpand\=\the\catcode`\=\relax
  \catcode`\noexpand\,\the\catcode`\,\relax
  \let\noexpand\XKVcatcodes\relax
}
\catcode`\=12\relax
\catcode`\,12\relax
\let\XKV@doxs\relax
```

`\XKV@warn` `\XKV@err` Warning and error macros.

```
\def\XKV@warn#1{\PackageWarning{xkeyval}{#1}}
\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.

```
\XKV@whilist\@filelist\XKV@tempa\ifx\XKV@documentclass\@undefined\fi{%
  \filename@parse\XKV@tempa
  \ifx\filename@ext\@clsextension
    \XKV@ifundefined{opt@\filename@base.\filename@ext
    }{}{\edef\XKV@documentclass{\filename@base.\filename@ext}}%
  \fi
}
```

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

```
\ifx\XKV@documentclass\@undefined
  \XKV@err{xkeyval loaded before \protect\documentclass}%
  \let\XKV@documentclass\@empty
  \let\XKV@classoptionslist\@empty
\else
  \let\XKV@classoptionslist\@classoptionslist
```

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

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

\XKV@testopte {⟨*function*⟩}
\XKV@t@stopte Macros for \ExecuteOptionsX and \ProcessOptionsX for testing for optional argu-
\XKV@t@st@pte ments and inserting default values. Execute ⟨*function*⟩ after preforming the checks.
\XKV@@t@st@pte

```
\def\XKV@testopte#1{%
  \XKV@ifstar{\XKV@sttrue\XKV@t@stopte#1}{\XKV@stfalse\XKV@t@stopte#1}%
}
\def\XKV@t@stopte#1{\@testopt{\XKV@t@st@pte#1}{KV}}
\def\XKV@t@st@pte#1[#2]{%
  \XKV@makepf{#2}%
  \@ifnextchar<{\XKV@@t@st@pte#1}%
    {\XKV@@t@st@pte#1<\@currname.\@currext>}%
}
\def\XKV@@t@st@pte#1<#2>{%
  \XKV@sp@deflist\XKV@fams{#2}%
  \@testopt#1{}%
}
```

Macros for class and package writers. These are mainly shortcuts to \define@key and \setkeys. The LaTeX macro \@fileswith@pti@ns 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.

```
\def\DeclareOptionX{%
  \let\@fileswith@pti@ns\@badrequireerror
  \XKV@ifstar\XKV@dox\XKV@d@x
}
```

\XKV@dox This macro defines \XKV@doxs to be used for unknown options.

```
\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

```
\def\XKV@d@x{\@testopt\XKV@@d@x{KV}}
\def\XKV@@d@x[#1]{%
  \@ifnextchar<{\XKV@@@d@x[#1]}{\XKV@@@d@x[#1]<\@currname.\@currext>}%
}
\def\XKV@@@d@x[#1]<#2>#3{\@testopt{\define@key[#1]{#2}{#3}}{}}
```

\ExecuteOptionsX [⟨*prefix*⟩]{⟨*families*⟩}[⟨*na*⟩]{⟨*key=value list*⟩}
This macro sets keys to specified values and uses \XKV@setkeys to do the job. Insert 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.

```
\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 submitted 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.

```
\def\ProcessOptionsX{\XKV@plfalse\XKV@testopte\XKV@pox}
```

`\XKV@pox` `[⟨na⟩]`

Workhorse for `\ProcessOptionsX` and `\ProcessOptionsX*`.

```
\def\XKV@pox[#1]{%
  \let\XKV@tempa\@empty
```

Set `\XKV@inpox`: indicates that we are in `\ProcessOptionsX` to invoke a special routine in `\XKV@s@tkeys`.

```
  \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`.

```
  \let\@fileswith@pti@ns\@badrequireerror
  \edef\XKV@testclass{\@currname.\@currext}%
```

If xkeyval is loaded by the document class, initialize `\@unusedoptionlist`.

```
  \ifx\XKV@testclass\XKV@documentclass
    \let\@unusedoptionlist\XKV@classoptionslist
    \XKV@ifundefined{ver@xkvltxp.sty}{}{%
      \@onelevel@sanitize\@unusedoptionlist
    }%
  \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.

```
    \ifXKV@st
      \def\XKV@tempb##1,{%
        \def\CurrentOption{##1}%
        \ifx\CurrentOption\@nnil\else
          \XKV@g@tkeyname##1=\@nil\CurrentOption
          \XKV@key@if@ndefined{\CurrentOption}{}{%
```

If the option also exists in local families, add it to the list for later use and remove it from `\@unusedoptionlist`.

```
            \XKV@useoption{##1}%
            \XKV@addtolist@n\XKV@tempa{##1}%
          }%
          \expandafter\XKV@tempb
        \fi
      }%
      \expandafter\XKV@tempb\XKV@classoptionslist,\@nil,%
    \fi
  \fi
```

Add current package options to the list.

```
  \expandafter\XKV@addtolist@o\expandafter
    \XKV@tempa\csname opt@\@currname.\@currext\endcsname
```

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

```
  \def\XKV@tempb{\XKV@setkeys[#1]}%
  \expandafter\XKV@tempb\expandafter{\XKV@tempa}%
```

Reset the macro created by `\DeclareOptionX*` to avoid processing future unknown keys using `\XKV@doxs`.

```
  \let\XKV@doxs\relax
```

Reset the \XKV@rm macro to avoid processing remaining options with \setrmkeys.

```
  \let\XKV@rm\@empty
```

Reset \ifXKV@inpox: not in \ProcessOptionsX anymore.

```
  \XKV@inpoxfalse
```

Reset \@fileswith@pti@ns to allow loading of classes or packages again.

```
  \let\@fileswith@pti@ns\@@fileswith@pti@ns
  \AtEndOfPackage{\let\@unprocessedoptions\relax}%
}
```

\XKV@useoption {⟨*option*⟩}

Removes an option from \@unusedoptionlist.

```
\def\XKV@useoption#1{%
  \def\XKV@resa{#1}%
  \XKV@ifundefined{ver@xkvltxp.sty}{}{%
    \@onelevel@sanitize\XKV@resa
  }%
  \@expandtwoargs\@removeelement{\XKV@resa}%
    {\@unusedoptionlist}\@unusedoptionlist
}
```

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

```
\DeclareOptionX*{%
  \PackageWarning{xkeyval}{Unknown option `\CurrentOption'}%
}
\ProcessOptionsX
```

Reset catcodes.

```
\XKVcatcodes
⟨/xkvlatex⟩
```

## 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.

```
%<*xkvkeyval>
%%
%% Based on keyval.sty.
%%
\def\XKV@tempa#1{%
\long\def\KV@@sp@def##1##2{%
  \futurelet\XKV@resa\KV@@sp@d##2\@nil\@nil#1\@nil\relax##1}%
\long\def\KV@@sp@d{%
  \ifx\XKV@resa\@sptoken
    \expandafter\KV@@sp@b
  \else
    \expandafter\KV@@sp@b\expandafter#1%
  \fi}%
```

```
\long\def\KV@@sp@b#1##1 \@nil{\KV@@sp@c##1}%
 }
\XKV@tempa{ }
\long\def\KV@@sp@c#1\@nil#2\relax#3{\XKV@toks{#1}\edef#3{\the\XKV@toks}}
\long\def\KV@do#1,{%
 \ifx\relax#1\@empty\else
  \KV@split#1==\relax
  \expandafter\KV@do\fi}
\long\def\KV@split#1=#2=#3\relax{%
  \KV@@sp@def\XKV@tempa{#1}%
  \ifx\XKV@tempa\@empty\else
    \expandafter\let\expandafter\XKV@tempc
      \csname\KV@prefix\XKV@tempa\endcsname
    \ifx\XKV@tempc\relax
      \XKV@err{‘\XKV@tempa’ undefined}%
    \else
      \ifx\@empty#3\@empty
        \KV@default
      \else
        \KV@@sp@def\XKV@tempb{#2}%
        \expandafter\XKV@tempc\expandafter{\XKV@tempb}\relax
      \fi
    \fi
  \fi}
\def\KV@default{%
  \expandafter\let\expandafter\XKV@tempb
    \csname\KV@prefix\XKV@tempa @default\endcsname
  \ifx\XKV@tempb\relax
    \XKV@err{No value specified for key ‘\XKV@tempa’}%
  \else
    \XKV@tempb\relax
  \fi}
\def\KV@def#1#2[#3]{%
  \long\@namedef{KV@#1@#2@default\expandafter}\expandafter
    {\csname KV@#1@#2\endcsname{#3}}%
  \long\@namedef{KV@#1@#2}##1}
```
⟨/xkvkeyval⟩

## 14.4 xkvtxhdr.tex

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

```
%<*xkvheader>
%%
%% Taken from latex.ltx.
%%
\message{2005/02/22 v1.1 xkeyval TeX header (HA)}
\def\@nnil{\@nil}
\def\@empty{}
\def\newif#1{%
  \count@\escapechar \escapechar\m@ne
    \let#1\iffalse
    \@if#1\iftrue
```

```
    \@if#1\iffalse
  \escapechar\count@}
\def\@if#1#2{%
  \expandafter\def\csname\expandafter\@gobbletwo\string#1%
                      \expandafter\@gobbletwo\string#2\endcsname
                         {\let#1#2}}
\long\def\@ifnextchar#1#2#3{%
  \let\reserved@d=#1%
  \def\reserved@a{#2}%
  \def\reserved@b{#3}%
  \futurelet\@let@token\@ifnch}
\def\@ifnch{%
  \ifx\@let@token\@sptoken
    \let\reserved@c\@xifnch
  \else
    \ifx\@let@token\reserved@d
      \let\reserved@c\reserved@a
    \else
      \let\reserved@c\reserved@b
    \fi
  \fi
  \reserved@c}
\def\:{\let\@sptoken= } \:  % this makes \@sptoken a space token
\def\:{\@xifnch} \expandafter\def\: {\futurelet\@let@token\@ifnch}
\let\kernel@ifnextchar\@ifnextchar
\long\def\@testopt#1#2{%
  \kernel@ifnextchar[{#1}{#1[{#2}]}}
\long\def\@firstofone#1{#1}
\long\def \@gobble #1{}
\long\def \@gobbletwo #1#2{}
\def\@expandtwoargs#1#2#3{%
\edef\reserved@a{\noexpand#1{#2}{#3}}\reserved@a}
\edef\@backslashchar{\expandafter\@gobble\string\\}
\newif\ifin@
\def\in@#1#2{%
 \def\in@@##1#1##2##3\in@@{%
  \ifx\in@##2\in@false\else\in@true\fi}%
 \in@@#2#1\in@\in@@}
\def\strip@prefix#1>{}
\def \@onelevel@sanitize #1{%
  \edef #1{\expandafter\strip@prefix
            \meaning #1}%
}
```
⟨/xkvheader⟩

## 14.5 xkvutils.tex

Avoid loading xkvutils.tex twice.

```
%<*xkvutils>
\csname XKeyValUtilsLoaded\endcsname
\let\XKeyValUtilsLoaded\endinput
\edef\XKeyValUtilsCatcodes{%
  \catcode`\noexpand\@\the\catcode`\@\relax
  \let\noexpand\XKeyValUtilsCatcodes\relax
```

```
}
\catcode`\@=11\relax
```

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).

```
\newtoks\XKV@toks
\newtoks\XKV@tempa@toks
```

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

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

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

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

`\XKV@ifundefined` {⟨*csname*⟩}{⟨*undefined*⟩}{⟨*defined*⟩}
Executes ⟨*undefined*⟩ if the control sequence with name ⟨*csname*⟩ is undefined, else it executes ⟨*defined*⟩. This macro uses ε-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 ⟨*undefined*⟩ if the control sequence is undefined or `\relax`, whereas the second version will only execute ⟨*undefined*⟩ if the control sequence is undefined. This is no problem for the applications in this package.

```
\ifx\ifcsname\@undefined\XKV@afterelsefi
  \def\XKV@ifundefined#1{%
    \begingroup\expandafter\expandafter\expandafter\endgroup
      \expandafter\ifx\csname#1\endcsname\relax
      \expandafter\@firstoftwo
    \else
      \expandafter\@secondoftwo
    \fi
  }
\else
  \def\XKV@ifundefined#1{%
    \ifcsname#1\endcsname
      \expandafter\@secondoftwo
    \else
      \expandafter\@firstoftwo
    \fi
  }
\fi
```

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

```
\XKV@ifundefined{ver@keyval.sty}{
  \input keyval
  \expandafter\def\csname ver@keyval.sty\endcsname{1999/03/16}
}{}
```

`\@ifnextcharacter` `\@ifncharacter` 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.
Contributed by Donald Arseneau.

```
\long\def\@ifnextcharacter#1#2#3{%
  \@ifnextchar\bgroup
  {\@ifnextchar{#1}{#2}{#3}}%
  {\@ifncharacter{#1}{#2}{#3}}%
}
\long\def\@ifncharacter#1#2#3#4{%
  \if\string#1\string#4%
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {#2}{#3}#4%
}
```

`\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.

```
\long\def\XKV@for@n#1#2#3{%
  \XKV@tempa@toks{#1}\edef#2{\the\XKV@tempa@toks}%
  \ifx#2\@empty
    \XKV@for@break
  \else
    \expandafter\XKV@f@r
  \fi
  #2{#3}#1,\@nil,%
}
```

`\XKV@f@r` ⟨*cmd*⟩{⟨*function*⟩}⟨*entry*⟩,
Looping macro.

```
\long\def\XKV@f@r#1#2#3,{%
  \XKV@tempa@toks{#3}\edef#1{\the\XKV@tempa@toks}%
  \ifx#1\@nnil
    \expandafter\@gobbletwo
  \else
    #2\expandafter\XKV@f@r
  \fi
  #1{#2}%
}
```

`\XKV@for@break` ⟨*text*⟩`\@nil`,
Macro to stop the for-loop.

```
\long\def\XKV@for@break #1\@nil,{\fi}
```

`\XKV@for@o` ⟨*listcmd*⟩⟨*cmd*⟩{⟨*function*⟩}
⟨*listcmd*⟩ is expanded once before starting the loop.

```
\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.

```
\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.

```
\long\def\XKV@for@eo#1#2#3{%
  \def#2{\XKV@f@r#2{#3}}\expandafter#2#1,\@nil,%
}
```

`\XKV@whilist` ⟨*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.

```
\long\def\XKV@whilist#1#2#3\fi#4{%
```

Check whether the condition is true and start iteration.

```
  #3\expandafter\XKV@wh@list#1,\@nil,\@nil\@@#2#3\fi{#4}{}\fi
}
```

`\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.

```
\long\def\XKV@wh@list#1,#2\@@#3#4\fi#5#6{%
```

Define the running ⟨*cmd*⟩.

```
  \def#3{#1}%
```

If we find the end of the list, stop.

```
  \ifx#3\@nnil
    \def#3{#6}\expandafter\XKV@wh@l@st
  \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.

```
    #4%
      #5\expandafter\expandafter\expandafter\XKV@wh@list
    \else
      \def#3{#6}\expandafter\expandafter\expandafter\XKV@wh@l@st
    \fi
  \fi
  #2\@@#3#4\fi{#5}{#1}%
}
```

`\XKV@wh@l@st` ⟨*text*⟩`\@@`⟨*cmd*⟩⟨*if*⟩`\fi{`⟨*function*⟩`}{`⟨*previous*⟩`}`
Macro to gobble remaining input.

```
\long\def\XKV@wh@l@st#1\@@#2#3\fi#4#5{}
```

`\XKV@addtomacro@n` ⟨*macro*⟩`{`⟨*content*⟩`}`
Adds ⟨*content*⟩ to ⟨*macro*⟩ without expanding it.

```
\long\def\XKV@addtomacro@n#1#2{%
  \XKV@tempa@toks\expandafter{#1#2}%
  \edef#1{\the\XKV@tempa@toks}%
}
```

`\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.

```
\def\XKV@addtomacro@o#1#2{%
  \expandafter\XKV@addtomacro@n\expandafter#1\expandafter{#2}%
}
```

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

```
\def\XKV@addtolist@n#1#2{%
  \ifx#1\@empty
    \XKV@addtomacro@n#1{#2}%
  \else
    \XKV@addtomacro@n#1{,#2}%
  \fi
}
```

`\XKV@addtolist@o` ⟨*cmd*⟩`{`⟨*content*⟩`}`
Adds ⟨*content*⟩ to the list in ⟨*cmd*⟩ after expanding the first token in ⟨*content*⟩ once.

```
\def\XKV@addtolist@o#1#2{%
  \ifx#1\@empty
    \XKV@addtomacro@o#1#2%
  \else
    \XKV@addtomacro@o#1{\expandafter,#2}%
  \fi
}
```

`\XKV@addtolist@x` ⟨*cmd*⟩`{`⟨*content*⟩`}`
Adds ⟨*content*⟩ to the list in ⟨*cmd*⟩ after a full expansion of both ⟨*cmd*⟩ and ⟨*content*⟩.

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

`\@selective@sanitize` `[`⟨*level*⟩`]{`⟨*character string*⟩`}{`⟨*cmd*⟩`}`
`\@s@lective@sanitize`
Converts selected characters, given by ⟨*character string*⟩, within the first-level expansion of ⟨*cmd*⟩ to category code 12, leaving all other tokens (including grouping braces) untouched. Thus, macros inside ⟨*cmd*⟩ do not lose their function, as it is the case with `\@onelevel@sanitize`. The resulting token list is again saved in ⟨*cmd*⟩.
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: ⟨*cmd*⟩ should not contain the control sequence `\bgroup`; however, `\csname`

bgroup\endcsname and \egroup are possible. The optional ⟨*level*⟩ 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.

```
\def\@selective@sanitize{\@testopt\@s@lective@sanitize\@M}
\def\@s@lective@sanitize[#1]#2#3{%
  \begingroup
    \count@#1\relax\advance\count@\@ne
    \XKV@toks\expandafter{#3}%
    \def#3{#2}\@onelevel@sanitize#3%
    \edef#3{{#3}{\the\XKV@toks}}%
    \expandafter\@s@l@ctive@sanitize\expandafter#3#3%
    \expandafter\XKV@tempa@toks\expandafter{#3}%
  \expandafter\endgroup\expandafter\toks@\expandafter{\the\XKV@tempa@toks}%
  \edef#3{\the\toks@}%
}
```

`\@s@l@ctive@sanitize` {⟨*cmd*⟩}{⟨*sanitized character string*⟩}{⟨*token list*⟩}

Performs the main work. Here, the characters in ⟨*sanitized character string*⟩ are already converted to catcode 12, ⟨*token list*⟩ is the first-level expansion of the original contents of ⟨*cmd*⟩. The macro basically steps through the ⟨*token list*⟩, 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 TEX primitives in general – which are characterized by a \meaning that starts with a backslash – than to test whether a token equals specifically \if, \else, \fi, etc. Note that \@s@l@ctive@sanitize is being called recursively if ⟨*token list*⟩ contains grouping braces.

```
\def\@s@l@ctive@sanitize#1#2#3{%
  \def\@i{\futurelet\@@tok\@ii}%
  \def\@ii{%
    \expandafter\@iii\meaning\@@tok\relax
    \ifx\@@tok\@s@l@ctive@sanitize
      \let\@@cmd\@gobble
    \else
      \ifx\@@tok\@sptoken
        \XKV@toks\expandafter{#1}\edef#1{\the\XKV@toks\space}%
        \def\@@cmd{\afterassignment\@i\let\@@tok= }%
      \else
        \let\@@cmd\@iv
      \fi
    \fi
    \@@cmd
  }%
  \def\@iii##1##2\relax{\if##1\@backslashchar\let\@@tok\relax\fi}%
  \def\@iv##1{%
    \toks@\expandafter{#1}\XKV@toks{##1}%
    \ifx\@@tok\bgroup
      \advance\count@\m@ne
      \ifnum\count@>\z@
        \begingroup
          \def#1{\expandafter\@s@l@ctive@sanitize
            \csname\string#1\endcsname{#2}}%
```

```
          \expandafter#1\expandafter{\the\XKV@toks}%
          \XKV@toks\expandafter\expandafter\expandafter
            {\csname\string#1\endcsname}%
          \edef#1{\noexpand\XKV@toks{\the\XKV@toks}}%
        \expandafter\endgroup#1%
      \fi
      \edef#1{\the\toks@{\the\XKV@toks}}%
      \advance\count@\@ne
      \let\@@cmd\@i
    \else
      \edef#1{\expandafter\string\the\XKV@toks}%
      \expandafter\in@\expandafter{#1}{#2}%
      \edef#1{\the\toks@\ifin@#1\else
              \ifx\@@tok\@sptoken\space\else\the\XKV@toks\fi\fi}%
      \edef\@@cmd{\noexpand\@i\ifx\@@tok\@sptoken\the\XKV@toks\fi}%
    \fi
    \@@cmd
  }%
  \let#1\@empty\@i#3\@s@l@ctive@sanitize
}
```

`\XKV@checksanitizea` {⟨*content*⟩}⟨*cmd*⟩

Check whether ⟨*content*⟩, to be saved to macro ⟨*cmd*⟩ unexpanded, contains the characters = or , with wrong catcodes. If so, it sanitizes them before saving ⟨*content*⟩ to ⟨*cmd*⟩.

```
\long\def\XKV@checksanitizea#1#2{%
  \XKV@ch@cksanitize{#1}#2=%
  \ifin@\else\XKV@ch@cksanitize{#1}#2,\fi
  \ifin@\@selective@sanitize[0]{,=}#2\fi
}
```

`\XKV@checksanitizeb` {⟨*content*⟩}⟨*cmd*⟩

Similar to `\XKV@checksanitizea`, but only checks commas.

```
\def\XKV@checksanitizeb#1#2{%
  \XKV@ch@cksanitize{#1}#2,%
  \ifin@\@selective@sanitize[0],#2\fi
}
```

`\XKV@ch@cksanitize` {⟨*character string*⟩}⟨*cmd*⟩⟨*token*⟩

This macro first checks whether at least one ⟨*token*⟩ is in ⟨*character string*⟩. 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 ⟨*token*⟩ (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 ⟨*token*⟩ in ⟨*character string*⟩ have the same catcode. ⟨*cmd*⟩ is used as a temporary macro and will contain ⟨*character string*⟩ at the end of the macro.

```
\long\def\XKV@ch@cksanitize#1#2#3{%
  \XKV@tempa@toks{#1}\edef#2{\the\XKV@tempa@toks}%
  \@onelevel@sanitize#2%
```

Check whether there is at least one = present.

```
  \@expandtwoargs\in@#3{#2}%
  \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 TEX before (and for instance stored in a macro or token register).

```
    \long\def#2##1#3##2\@nil{%
      \XKV@tempa@toks{##2}\edef#2{\the\XKV@tempa@toks}%
      \ifx#2\@empty\else\in@false\fi
    }%
    #2#1#3\@nil
  \fi
  \XKV@tempa@toks{#1}\edef#2{\the\XKV@tempa@toks}%
}
```

`\XKV@sp@deflist` ⟨*cmd*⟩{⟨*token list*⟩}

Defines ⟨*cmd*⟩ as ⟨*token list*⟩ after removing spaces surrounding elements of the list in ⟨*token list*⟩. So, `keya, key b` becomes `keya,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.

```
\def\XKV@sp@deflist#1#2{%
  \let#1\@empty
  \XKV@for@n{#2}\XKV@resa{%
    \expandafter\KV@@sp@def\expandafter\XKV@resa\expandafter{\XKV@resa}%
    \XKV@addtomacro@o#1{\expandafter,\XKV@resa}%
  }%
  \ifx#1\@empty\else
    \def\XKV@resa,##1\@nil{\def#1{##1}}%
    \expandafter\XKV@resa#1\@nil
  \fi
}
```

`\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 particulary 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.

```
\def\XKV@merge#1#2#3{%
  \XKV@checksanitizea{#2}\XKV@tempa
```

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

```
  \XKV@for@o\XKV@tempa\XKV@tempa{%
    \XKV@pltrue
```

Retrieve the key name of the new item at hand.

```
    #3\XKV@tempa\XKV@tempb
```

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

```
    \let\XKV@tempc#1%
    \let#1\@empty
```

Start a loop over the old list.

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

Retrieve the key name of the old key at hand.

```
      #3\XKV@tempc\XKV@tempd
      \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.

```
        \XKV@plfalse
        \XKV@addtolist@o#1\XKV@tempa
      \else
```

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

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

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

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

If requested, save the new list globally.

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

`\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`.

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

Sanitize comma's.

```
  \XKV@checksanitizeb{#2}\XKV@tempa
```

Copy the current list and make the original empty.

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

Run over the current list.

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

Get the key name to identify the current entry.

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

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

```
    \@expandtwoargs\in@{,\XKV@tempc,}{,\XKV@tempa,}%
    \ifin@\else\XKV@addtolist@o#1\XKV@tempb\fi
  }%
```

Save globally is necessary.

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

Finalize.

```
\XKeyValUtilsCatcodes
⟨/xkvutils⟩
```

## 14.6 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.

```
%<*xkvview>
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesPackage{xkvview}%
  [2008/08/10 v1.5 viewer utility for xkeyval (HA)]
\RequirePackage{xkeyval}
\RequirePackage{longtable}
\DeclareOptionX*{%
  \PackageWarning{xkvview}{Unknown option `\CurrentOption'}%
}
\ProcessOptionsX
```

Initializations.

```
\newif\ifXKVV@vwkey
\newif\ifXKVV@colii
\newif\ifXKVV@coliii
\newif\ifXKVV@coliv
\newif\ifXKVV@colv
\newwrite\XKVV@out
\let\XKVV@db\@empty
```

Setup options and presets.

```
\define@cmdkeys[XKVV]{xkvview}[XKVV@]{%
  prefix,family,type,default,file,columns,wcolsep,weol}[\@nil]
\define@boolkeys[XKVV]{xkvview}[XKVV@]{view,vlabels,wlabels}[true]
\presetkeys[XKVV]{xkvview}{prefix,family,type,default,file,%
  columns,wcolsep=&,weol=\\,view,vlabels=false,wlabels=false}{}
```

`\XKVV@tabulate` `\XKVV@t@bulate`

{⟨*key*⟩}{⟨*type*⟩}{⟨*default*⟩}

Adds the input information to the main database in `\XKVV@db`.

```
\def\XKVV@tabulate#1#2#3{%
  \def\XKV@tempa{#3}%
  \@onelevel@sanitize\XKV@tempa
  \XKV@addtolist@x\XKVV@db{#1=\ifx\XKV@prefix\@empty\else\expandafter
    \XKVV@t@bulate\XKV@prefix\fi=\XKV@tfam=#2=\XKV@tempa}%
}
\def\XKVV@t@bulate#1@{#1}
```

`\XKV@define@key` `\XKV@d@fine@k@y` `\XKV@define@cmdkey` `\XKV@d@fine@ch@icekey` `\XKV@d@fine@ch@ic@key` `\XKV@d@f@ne@b@olkey`

Redefine the internals of key defining macros to record information in the database.

```
\def\XKV@define@key#1{%
  \@ifnextchar[{\XKV@d@fine@k@y{#1}}{%
    \XKVV@tabulate{#1}{ordinary}{[none]}%
    \expandafter\def\csname\XKV@header#1\endcsname####1%
  }%
```

```
}
\def\XKV@d@fine@k@y#1[#2]{%
  \XKVV@tabulate{#1}{ordinary}{#2}%
  \XKV@define@default{#1}{#2}%
  \expandafter\def\csname\XKV@header#1\endcsname##1%
}
\def\XKV@define@cmdkey#1#2[#3]#4{%
  \ifXKV@st
    \XKVV@tabulate{#2}{command}{#3}%
    \XKV@define@default{#2}{#3}%
  \else
    \XKVV@tabulate{#2}{command}{[none]}%
  \fi
  \def\XKV@tempa{\expandafter\def\csname\XKV@header#2\endcsname####1}%
  \begingroup\expandafter\endgroup\expandafter\XKV@tempa\expandafter
    {\expandafter\def\csname#1#2\endcsname{##1}#4}%
}
\def\XKV@d@fine@ch@icekey#1[#2]{%
  \XKVV@tabulate{#1}{choice}{#2}%
  \XKV@define@default{#1}{#2}%
  \XKV@d@fine@ch@ic@key{#1}%
}
\def\XKV@d@fine@ch@ic@key#1{%
  \XKVV@tabulate{#1}{choice}{[none]}%
  \ifXKV@pl\XKV@afterelsefi
    \expandafter\XKV@d@f@ne@ch@ic@k@y
  \else\XKV@afterfi
    \expandafter\XKV@d@f@ne@ch@ic@key
  \fi
  \csname\XKV@header#1\endcsname
}
\def\XKV@d@f@ne@b@olkey#1#2#3#4#5{%
  \expandafter\newif\csname if#3\endcsname
  \ifXKV@st
    \XKVV@tabulate{#2}{boolean}{#4}%
    \XKV@define@default{#2}{#4}%
  \else
    \XKVV@tabulate{#2}{boolean}{[none]}%
  \fi
  \ifXKV@pl
    \def#1##1{\XKV@pltrue\XKV@sttrue
      \XKV@checkchoice[\XKV@resa]{##1}{true,false}#5%
    }%
  \else
    \def#1##1{\XKV@plfalse\XKV@sttrue
      \XKV@checkchoice[\XKV@resa]{##1}{true,false}#5%
    }%
  \fi
}
```

`\xkvview` {⟨*options*⟩}

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

```
\def\xkvview#1{%
```

Process all options.

```
  \setkeys[XKVV]{xkvview}{#1}%
  \ifx\XKVV@default\@nnil\else\@onelevel@sanitize\XKVV@default\fi
```

If no column information, display all columns.

```
  \ifx\XKVV@columns\@nnil
    \count@5
    \XKVV@coliitrue\XKVV@coliiitrue\XKVV@colivtrue\XKVV@colvtrue
  \else
```

Check how much and which columns should be displayed.

```
    \count@\@ne
    \@expandtwoargs\in@{,prefix,}{,\XKVV@columns,}%
    \ifin@\advance\count@\@ne\XKVV@coliitrue\else\XKVV@coliifalse\fi
    \@expandtwoargs\in@{,family,}{,\XKVV@columns,}%
    \ifin@\advance\count@\@ne\XKVV@coliiitrue\else\XKVV@coliiifalse\fi
    \@expandtwoargs\in@{,type,}{,\XKVV@columns,}%
    \ifin@\advance\count@\@ne\XKVV@colivtrue\else\XKVV@colivfalse\fi
    \@expandtwoargs\in@{,default,}{,\XKVV@columns,}%
    \ifin@\advance\count@\@ne\XKVV@colvtrue\else\XKVV@colvfalse\fi
  \fi
  \ifXKVV@view
```

Construct long table header.

```
    \protected@edef\XKV@tempa{\noexpand\begin{longtable}[l]{%
      *\the\count@ l}\normalfont Key\ifXKVV@colii&\normalfont Prefix%
      \fi\ifXKVV@coliii&\normalfont Family\fi\ifXKVV@coliv&\normalfont
      Type\fi\ifXKVV@colv&\normalfont Default\fi\\\noexpand\hline
      \noexpand\endfirsthead\noexpand\multicolumn{\the\count@}{l}{%
      \normalfont\emph{Continued from previous page}}\\\noexpand\hline
      \normalfont Key\ifXKVV@colii&\normalfont Prefix\fi\ifXKVV@coliii
      &\normalfont Family\fi\ifXKVV@coliv&\normalfont Type\fi
      \ifXKVV@colv&\normalfont Default\fi\\\noexpand\hline\noexpand
      \endhead\noexpand\hline\noexpand\multicolumn{\the\count@}{r}{%
      \normalfont\emph{Continued on next page}}\\\noexpand\endfoot
      \noexpand\hline\noexpand\endlastfoot
    }%
    \XKV@toks\expandafter{\XKV@tempa}%
  \fi
```

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

```
  \ifx\XKVV@file\@nnil\else\immediate\openout\XKVV@out\XKVV@file\fi
```

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

```
  \XKV@for@o\XKVV@db\XKV@tempa{%
    \XKVV@vwkeytrue\expandafter\XKVV@xkvview\XKV@tempa\@nil
  }%
```

Finish the long table and typeset it.

```
  \ifXKVV@view
    \addto@hook\XKV@toks{\end{longtable}}%
    \begingroup\ttfamily\the\XKV@toks\endgroup
  \fi
```

Close the target file.

```
  \ifx\XKVV@file\@nnil\else\immediate\closeout\XKVV@out\fi
}
```

`\XKVV@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.

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

Check whether the current entry satisfies all criteria.

```
  \ifx\XKVV@prefix\@nnil\else
    \def\XKV@tempa{#2}%
    \ifx\XKV@tempa\XKVV@prefix\else\XKVV@vwkeyfalse\fi
  \fi
  \ifx\XKVV@family\@nnil\else
    \def\XKV@tempa{#3}%
    \ifx\XKV@tempa\XKVV@family\else\XKVV@vwkeyfalse\fi
  \fi
  \ifx\XKVV@type\@nnil\else
    \def\XKV@tempa{#4}%
    \ifx\XKV@tempa\XKVV@type\else\XKVV@vwkeyfalse\fi
  \fi
  \ifx\XKVV@default\@nnil\else
    \def\XKV@tempa{#5}%
    \ifx\XKV@tempa\XKVV@default\else\XKVV@vwkeyfalse\fi
  \fi
  \ifXKVV@vwkey
```

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

```
    \ifXKVV@view
      \edef\XKV@tempa{%
        #1\ifXKVV@colii&#2\fi\ifXKVV@coliii&#3\fi
        \ifXKVV@coliv&#4\fi\ifXKVV@colv&#5\fi
        \ifXKVV@vlabels\noexpand\label{#2-#3-#1}\fi
      }%
      \expandafter\addto@hook\expandafter
        \XKV@toks\expandafter{\XKV@tempa\\}%
    \fi
    \ifx\XKVV@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.

```
      \immediate\write\XKVV@out{%
        #1\ifXKVV@colii\XKVV@wcolsep#2\fi
        \ifXKVV@coliii\XKVV@wcolsep#3\fi
        \ifXKVV@coliv\XKVV@wcolsep#4\fi
        \ifXKVV@colv\XKVV@wcolsep#5\fi
        \ifXKVV@wlabels\string\label{#2-#3-#1}\fi
        \expandafter\noexpand\XKVV@weol
      }%
    \fi
  \fi
}
⟨/xkvview⟩
```

## 14.7 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].

```
%<*xkvltxpatch>
%%
%% Based on latex.ltx.
%%
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesPackage{xkvltxp}[2014/05/25 v1.3 LaTeX2e kernel patch (HA)]
```

Load utilities needed in this package.

```
\input xkvutils
```

Start redefining internal LaTeX macros.

```
\def\@pass@ptions#1#2#3{%
  \def\reserved@a{#2}%
  \def\reserved@b{\CurrentOption}%
  \ifx\reserved@a\reserved@b
    \@ifundefined{opt@#3.#1}{\@temptokena\expandafter{#2}}{%
      \@temptokena\expandafter\expandafter\expandafter
        {\csname opt@#3.#1\endcsname}%
      \@temptokena\expandafter\expandafter\expandafter{%
        \expandafter\the\expandafter\@temptokena\expandafter,#2}%
    }%
  \else
    \@ifundefined{opt@#3.#1}{\@temptokena{#2}}{%
      \@temptokena\expandafter\expandafter\expandafter
        {\csname opt@#3.#1\endcsname}%
      \@temptokena\expandafter{\the\@temptokena,#2}%
    }%
  \fi
  \expandafter\xdef\csname opt@#3.#1\endcsname{\the\@temptokena}%
}
\def\OptionNotUsed{%
  \ifx\@currext\@clsextension
    \let\reserved@a\CurrentOption
    \@onelevel@sanitize\reserved@a
    \xdef\@unusedoptionlist{%
      \ifx\@unusedoptionlist\@empty\else\@unusedoptionlist,\fi
      \reserved@a}%
  \fi
}
\def\@use@ption{%
  \let\reserved@a\CurrentOption
  \@onelevel@sanitize\reserved@a
  \@expandtwoargs\@removeelement\reserved@a
  \@unusedoptionlist\@unusedoptionlist
  \csname ds@\CurrentOption\endcsname
}
```

\@fileswith@pti@ns

```
\def\@fileswith@pti@ns#1[#2]#3[#4]{%
  \XKV@sp@deflist\XKV@resb{#2}%
  \ifx#1\@clsextension
    \ifx\@classoptionslist\relax
      \let\@classoptionslist\XKV@resb
      \def\reserved@a{%
        \@onefilewithoptions#3[#2][#4]#1%
        \@documentclasshook}%
    \else
      \def\reserved@a{%
        \@onefilewithoptions#3[#2][#4]#1}%
    \fi
  \else
    \def\reserved@b##1,{%
      \ifx\@nil##1\relax\else
        \ifx\relax##1\relax\else
         \noexpand\@onefilewithoptions##1%
           [\XKV@resb][#4]\noexpand\@pkgextension
        \fi
        \expandafter\reserved@b
      \fi}%
      \edef\reserved@a{\zap@space#3 \@empty}%
      \edef\reserved@a{\expandafter\reserved@b\reserved@a,\@nil,}%
  \fi
  \reserved@a}

\let\@@fileswith@pti@ns\@fileswith@pti@ns
⟨/xkvltxpatch⟩
```

## 14.8 pst-xkey.tex

Avoid loading pst-xkey.tex twice.

```
%<*pxktex>
\csname PSTXKeyLoaded\endcsname
\let\PSTXKeyLoaded\endinput
\edef\PSTXKeyCatcodes{%
  \catcode`\noexpand\@\the\catcode`\@\relax
  \let\noexpand\PSTXKeyCatcodes\relax
}
\catcode`\@=11\relax
```

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

```
\ifx\ProvidesFile\@undefined
  \message{2005/11/25 v1.6 PSTricks specialization of xkeyval (HA)}
  \ifx\XKeyValLoaded\endinput\else\input xkeyval \fi
\else
  \ProvidesFile{pst-xkey.tex}
    [2005/11/25 v1.6 PSTricks specialization of xkeyval (HA)]
  \@addtofilelist{pst-xkey.tex}
  \RequirePackage{xkeyval}
\fi
```

\pst@famlist Initialize the list of families.

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

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

```
\def\pst@addfams#1{%
  \XKV@for@n{#1}\XKV@tempa{%
    \@expandtwoargs\in@{,\XKV@tempa,}{,\pst@famlist,}%
    \ifin@\else\edef\pst@famlist{\pst@famlist,\XKV@tempa}\fi
  }%
}
```

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

\pss@t

```
\def\psset{%
  \expandafter\@testopt\expandafter\pss@t\expandafter{\pst@famlist}%
}
\def\pss@t[#1]#2{\setkeys+[psset]{#1}{#2}\ignorespaces}
```

\@psset This macro defined by pstricks.tex is internally used as a shortcut. We have to redefine this as well to avoid problems.

```
\def\@psset#1,\@nil{%
  \edef\XKV@tempa{\noexpand\setkeys+[psset]{\pst@famlist}}%
  \XKV@tempa{#1}%
}
```

Finalize.

```
\PSTXKeyCatcodes
⟨/pxktex⟩
```

### 14.9 pst-xkey.sty

Initialize the package.

```
%<*pxklatex>
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesPackage{pst-xkey}
  [2005/11/25 v1.6 package wrapper for pst-xkey.tex (HA)]
```

Load required package.

```
\ifx\PSTXKeyLoaded\endinput\else\input pst-xkey \fi
```

Ignore options.

```
\DeclareOptionX*{%
  \PackageWarning{pst-xkey}{Unknown option '\CurrentOption'}%
}
\ProcessOptionsX
⟨/pxklatex⟩
```

## References


[1] Hendri Adriaens. extract package. CTAN:/macros/latex/contrib/extract.

[2] Johannes Braams, David Carlisle, Alan Jeffrey, Leslie Lamport, Frank Mittelbach, Chris Rowley, and Rainer Schöpf. The LaTeX2ε sources. CTAN:/macros/latex/base, 2003.

[3] David Carlisle. keyval package, v1.13, 1999/03/16. CTAN:/macros/latex/required/graphics.

[4] Frank Mittelbach and Michel Goossens. *The LaTeX Companion*. Tools and Techniques for Computer Typesetting. Addison-Wesley, Boston, Massachusetts, 2 edition, 2004. With Johannes Braams, David Carlisle, and Chris Rowley.

[5] Herbert Voß. PSTricks website. http://www.pstricks.de.

[6] Timothy Van Zandt et al. PSTricks package, v1.04, 2004/06/22. CTAN:/graphics/pstricks.

## Acknowledgements

The author is grateful to Carsten Heinz, Morten Høgholm, Josselin Noirel, Heiko Oberdiek, Will Robertson, Till Tantau, Herbert Voßand Joseph Wright for help and suggestions. Thanks go to Donald Arseneau for contributing the robust \@ifnextcharacter macro, to Morten Høgholm for contributing a fast for-loop macro and to David Carlisle for modifications to support \par. 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.

# Index

Numbers written in italic 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.