# The xkeyval package [*]

Hendri Adriaens
http://stuwww.uvt.nl/~hendri

v2.0 (2005/01/30)

**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.

## 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 default values. The same thing holds for the margins.

The idea is that one first defines some keys using the tools presented in section 2 in the preamble or in a package or class. These keys can perform some 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 3, 4 and 5. 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@cmdkey{mybox}{background}
\define@key{mybox}{left}{\setlength\myleft{#1}}
% and some other keys
\def\mybox{\@ifnextchar[\@mybox{\@mybox[]}}
\def\@mybox[#1]#2{%
  \setkeys{mybox}{#1}%
  % some operations
}
```

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 , keyd=end}
```

From values consisting entirely of a `{ }` group, the outer braces will be stripped off. 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, xkeyval will not terminate the `key=value` pair when it encounters a comma. For instance, see the value of `keyb` in the example above. The same story holds for the equality sign. Notice further that any white space around the characters `=` and `,` is ignored. Finally, `keyc` did not get a value. If no default value has been defined for this key, an error will be generated. More details can be found in sections 2, 3, 4 and 5.

Both keys defined using the keyval and xkeyval can be set by this package. The xkeyval macros allow for scanning multiple families for keys. This can, for example, be used to create local families for custom 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 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 basic keyval macros are supplied in `keyval.tex` to guarantee compatibility to packages that use those macros. Section 10 provides more information about this issue.

To load xkeyval, plain TEX users do `\input xkeyval`. LATEX users do either `\usepackage{xkeyval}` or `\RequirePackage{xkeyval}`. 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 with useless entries for instance when searching for keys in families.

PSTricks [5, 6] package authors should have a look at the pst-xkey package contained in the xkeyval package distribution [1] for an options system based on xkeyval, but which is specialized in defining and setting PSTricks keys.

The organization of this documentation is as follows. Section 2 will discuss the macros available to define keys. Section 3 will continue with describing the macros that can set the keys. Section 4 explains special syntax which will allow saving and copying key values. In section 5, the preset system will be introduced. Section 6 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. These will be discussed in section 7.1. An extension of the LATEX $2_\varepsilon$ kernel is discussed in section 7.2. This extension provides a way to use expandable macros in package options. Sections 9 and 10 discuss feedback that xkeyval might give and known issues, respectively.

Throughout this documentation, you will find some examples with a short description. More examples can be found in some example files that come with this package. See section 11 for more information. This section also provides the information how to generate the source code documentation from the source. This documentation provides the programming details of xkeyval.

## 2 Defining and checking keys

### 2.1 Ordinary keys

This section describes how to define ordinary keys.

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

This defines a macro `\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 list of options containing `key=value`, the macro `\prefix@family@key` receives `value` as its argument. 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. 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`.

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`.

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.[1]

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

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

## 2.2 Boolean keys

This section describes boolean keys which are either true or false. When comparing the macro of this section to `\define@key` of section 2.1, we see that the ⟨*function*⟩ is known (namely, set a conditional to true or false) and hence the macro has one mandatory argument less.

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

This creates a conditional of the form `\ifprefix@family@key`[2] using `\newif`[3] (which initiates the conditional to `\iffalse`) and a key macro of the form `\prefix@family@key` which is defined as `\XKV@setbool{prefix@family@key}{#1}`.

[1]Like PSTricks, which uses a system originating from keyval, but which has been modified to use no families and `psset` as prefix.

[2]When you want to use this macro directly, either make sure that neither of the input parameters contains 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.

[3]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 macro `\XKV@setbool` only takes the values `true` and `false` and uses that to set the conditional. The default value can only be `true` or `false` as well.

```
\define@boolkey{fam}{frame}
```

This example creates `\ifKV@fam@frame` and defines `\KV@fam@frame` to expand to `\XKV@setbool{KV@fam@frame}{#1}`.

## 2.3 Command keys

This section describes command keys. The macro described here is a specialized version of `\define@key` described in section 2.1 and ⟨*function*⟩ will store the user input in a macro.

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

This defines the key macro `\prefix@family@key`[2] with one argument to define a macro in the following way: `\def\prefix@family@key@cmd{#1}`.

```
\define@cmdkey{fam}{text}
```

This example defines the key `\KV@fam@text` to store user input to the `\setkeys` command (see section 3) in the macro `\KV@fam@key@text@cmd`.

## 2.4 Checking keys

This section provides a macro to check the existence of keys.

`\key@ifundefined[`⟨*prefix*⟩`]{`⟨*families*⟩`}{`⟨*key*⟩`}{`⟨*undefined*⟩`}{`⟨*defined*⟩`}`

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`.

## 2.5 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*⟩`}`

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.

## 3 Setting keys

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 2. This optional argument takes the value `KV` by default.

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

This macro sets keys of the form `\prefix@family@key` 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`. This is in line with how key macros are constructed (see section 2).

```
\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`. The next line will set `keyb` in `familya`. The last one sets `keyb` in family `familyb`.

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~and~\setkeys[my]{familya}{keya=#1}}
\setkeys[my]{familyb}{keyc=a\setkeys[my]{familya}{keya=~and~b}}
```

returns `a and b and a and b`.

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

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*`. 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⟩]
```

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⟩]
```

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⟩]
```

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. See also section 11 for more examples.

## 4 Pointers

The xkeyval package allows the use of pointers in key values. These 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.

### 4.1 Saving values

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

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

This example will set `keya` as usual, 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 4.2.

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⟩}
```

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⟩}
```

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.

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`.

## 4.2 Using saved values

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 2.1).

```
\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`.

## 5 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 2.1 in examples.

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

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`, section 4.1). The saved macros will be used by `\setkeys`, when they are defined, whenever ⟨*family*⟩ is used in the ⟨*families*⟩ argument of that macro. 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 macro works 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. Assuming that all keys in the next example have a default value, 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` contains `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*⟩`}`

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 deletion.

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

This command clears 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.

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.

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 example is 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 11.

## 6 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 that the keyval package is used inside a command.

For these reasons, xkeyval performs several actions with user input before trying to parse it. 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}
\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 7.1) are not protected as LaTeX does not protect them either.

## 7 xkeyval and LaTeX

If xkeyval is loaded by `\RequirePackage` or `\usepackage`, the package will 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 and stores that (including the class extension) into the following macro.

```
\XKV@documentclass
```

This macro could, for instance, contain `article.cls` and can be useful when using `\ProcessOptionsX*` in a class. See page 16.

Secondly, the global options stored in `\@classoptionslist` by LaTeX are copied to the following macro.

```
\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`.

## 7.1 Declaring and setting class or package options

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*⟩`}`

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.

`\DeclareOptionX*{`⟨*function*⟩`}`

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*⟩`}`

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*⟩`]`

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 explained in section 7.2. 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*⟩`]`

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.

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.

## 7.2 Options with macros

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. If you want to offer the user this functionality for the `\documentclass` command, the user will have to do `\RequirePackage{xkvltxp}` on the first line of the LaTeX file. If you plan to use 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 7.1) 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 11.

## 8 List of macro structures

This section provides a list of all reserved internal macros 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 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 through `\setkeys`.

`\ifpre@fam@key`, `\pre@fam@keytrue`, `\pre@fam@keyfalse`
: The conditional created by `\define@boolkey` with parameters `pre`, `fam` and `key`. The `true` and `false` macros are used to set the conditional to `\iftrue` and `\iffalse` respectively.

`\pre@fam@key@cmd`
: The macro to which input to `\setkeys` for `key` will be stored if this key has been defined by `\define@cmdkey`.

`\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 are defined to be used outside xkeyval macros as well. These are the macros `\ifpre@fam@key` and `\pre@fam@key@cmd`. To be able to use these macros yourself, none of the input parameters should contain a 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}
\define@cmdkey{some-fam}{mycmd}
\setkeys{some-fam}{myif=false,mycmd=save this}
```

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) `boolean can only be 'true' or 'false'` (error)
   A value other than `true` or `false` has been submitted to 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 4.2.

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 2.5.

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 4.1.

11) `no presets defined for 'pre@fam@'` (error)
    You are trying to delete or undefine presets that have not been defined yet. See section 5.

12) `xkeyval loaded before \documentclass` (error)
    Load xkeyval after `\documentclass` (or in the class that is the document class). See section 7.1.

## 10 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 [1], which is distributed with the xkeyval package. 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 using pst-key, which are pst-circ, pst-eucl, pst-fr3d, pst-geo, pst-gr3d, pst-labo, pst-lens, pst-ob3d, pst-optic, pst-osci, pst-poly, pst-stru, pst-uml and pst-vue3d.

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.

## 11 Source and examples

To generate the source code documentation, find the source of this package, `xkeyval.dtx` in your local TeX installation or on CTAN and 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
```

If you only want to produce the package and example files from the source, then the first step is sufficient. This step will generate the package files (`xkeyval.tex`, `xkeyval.sty`, `xkvltxp.sty`, `keyval.tex` and `xkvtxhdr.tex`) and the example files.

The file xkvex1.tex provides an example for TEX users for the macros described in sections 2, 3, 4 and 5. 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 7.1. The set of files consisting of xkvex4.tex, xkveca.cls, xkvecb.cls, xkvesa.sty, xkvesb.sty and xkvesc.sty provides an example for section 7.2. 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.

## 12 Implementation

### 12.1 TEX program

Avoid loading xkeyval.tex twice.

```
%<*tex>
\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
```

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

```
\newtoks\XKV@toks
\newif\ifXKV@st
\newif\ifXKV@sg
\newif\ifXKV@pl
\newif\ifXKV@knf
\newif\ifXKV@rkv
\newif\ifXKV@inpox
\let\XKV@rm\@empty
```

Load LATEX primitives if necessary and provide information.

```
\ifx\ProvidesFile\@undefined
  \message{2005/01/30 v2.0 key=value parser (HA)}
  \input xkvtxhdr.tex
\else
  \ProvidesFile{xkeyval.tex}[2005/01/30 v2.0 key=value parser (HA)]
  \@addtofilelist{xkeyval.tex}
\fi
```

`\@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 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 $\varepsilon$-TEX if possible to avoid filling TEX's hash when checking control sequences like key macros in the rest of the package. `\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.tex
  \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@whilist` ⟨*cmd*⟩`:=`⟨*list*⟩`\do`⟨*if*⟩`\fi{`⟨*function*⟩`}`

Based on `\XKV@for`. 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`, but contains an additional check and is hence less efficient than `\XKV@for` in that situation.

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

Check whether the condition is true and start iteration.

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

`\XKV@wh@list` Performs iteration and checks extra condition. This macro is not optimized for the case that the list contains a single element.

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

Define the running ⟨*cmd*⟩ in a group to keep it local in case we have to exit the loop.

```
  \begingroup\def#3{#1}\expandafter\endgroup
```

If we find the end of the list, stop.

```
  \ifx#3\@nnil
    \expandafter\XKV@wh@l@st
  \else
```

If the condition is met, define the running ⟨*cmd*⟩, execute ⟨*function*⟩ and continue. Otherwise stop.

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

`\XKV@wh@l@st` Macro to gobble remaining input.

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

`\XKV@for` ⟨*cmd*⟩`:=`⟨*list*⟩`\do{`⟨*function*⟩`}`

Based on `\@for`, but also starts execution of ⟨*function*⟩ when ⟨*list*⟩ is empty. This is done to support packages that use the ‘empty family’, like PSTricks. The macro executes ⟨*function*⟩ while ⟨*if*⟩ is valid. At every iteration, the first element will be taken from ⟨*list*⟩ and ⟨*cmd*⟩ will be defined to expand to this element. Execution stops when the list has ran out of elements.

```
\long\def\XKV@for#1:=#2\do#3{\expandafter\@forloop#2,\@nil,\@nil\@@#1{#3}}
```

`\XKV@addtomacro@n` {⟨*macro*⟩}{⟨*content*⟩}

Adds ⟨*content*⟩ to ⟨*macro*⟩ without expanding it.

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

`\XKV@addtomacro@o` {⟨*macro*⟩}{⟨*content*⟩}

Adds ⟨*content*⟩ to ⟨*macro*⟩ after expanding the first token of ⟨*content*⟩ once.

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

`\XKV@addtolist@n` {⟨*cmd*⟩}{⟨*token*⟩}

Adds ⟨*token*⟩ to the list in ⟨*cmd*⟩ without expanding ⟨*token*⟩.

```
\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*⟩}{⟨*token*⟩}

Adds ⟨*token*⟩ to the list in ⟨*cmd*⟩ after expanding ⟨*token*⟩ 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*⟩}{⟨*token*⟩}

Adds ⟨*token*⟩ to the list in ⟨*cmd*⟩ after a full expansion of both ⟨*cmd*⟩ and ⟨*token*⟩.

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

`\@selective@sanitize`
`\@s@lective@sanitize` [⟨*level*⟩]{⟨*character string*⟩}{⟨*cmd*⟩}

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\endgroup\expandafter\def\expandafter#3\expandafter{#3}%
}
```

`\@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` Check whether the content `#1`, to be saved to macro `#2` contains the characters `=` or `,` with wrong catcodes. If so, sanitize them.

```
\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` 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*⟩`}{`⟨*token list*⟩`}{`⟨*token*⟩`}`
This macro first check 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 value. But we won't worry about those characters anyway since the relevant key macro will have to process that.

```
\def\XKV@ch@cksanitize#1#2#3{%
  \def#2{#1}%
  \@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).

```
    \def#2##1#3##2\@nil{%
      \def#2{##2}%
      \ifx#2\@empty\else\in@false\fi
    }%
    #2#1#3\@nil
```

```
  \fi
  \def#2{#1}%
}
```

`\XKV@merge` 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 macro in argument `#3` is used to filter the key name from the syntax. 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 macro `#3` is `\XKV@getsg` and `\XKV@getkeyname` respectively.

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

We have to do merging. Start the loop over the new presets. At every iteration, one new preset will be checked with old presets.

```
  \@for\XKV@tempa:=\XKV@tempa\do{%
    \XKV@pltrue
```

Retrieve the key name of the new preset at hand.

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

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

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

Start a loop over the old values.

```
    \@for\XKV@tempc:=\XKV@tempc\do{%
```

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 preset 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 preset to the list.

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

If, after checking the old presets, no old preset has been overwritten then append the new preset to the end of the 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` Delete entries by key name from a list of presets or save keys. `#1` is the macro currently holding the list to be updated. `#2` is the list of entries that should be removed and `#3` is the macro that should be used to retrieve the key name from an entry. For `\delpresetkeys` this is `\XKV@getkeyname` and for `\delsavekeys` it is `\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.

```
  \@for\XKV@tempb:=\XKV@tempb\do{%
```

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
}
```

`\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@sp@deflist` `{`⟨*cmd*⟩`}{`⟨*token*⟩`}`
Defines ⟨*cmd*⟩ as ⟨*token*⟩ after removing spaces surrounding elements of the list in ⟨*token*⟩. 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
  \def\XKV@resa{#2}%
  \@for\XKV@resa:=\XKV@resa\do{%
    \expandafter\KV@@sp@def\expandafter\XKV@resa\expandafter{\XKV@resa}%
    \edef#1{#1,\XKV@resa}%
  }%
  \ifx#1\@empty\else
    \def\XKV@resa,##1\@nil{\def#1{##1}}%
    \expandafter\XKV@resa#1\@nil
  \fi
}
```

`\XKV@makepf` This macro creates the prefix, like `prefix@` in `\prefix@family@key`. First it deletes spaces from the input and checks wether 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
    \ifx\XKV@prefix\@empty\else
      \edef\XKV@prefix{\XKV@prefix @}%
    \fi
  \fi
}
```

`\XKV@makehd` 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}%
  \ifx\XKV@header\@empty
    \let\XKV@header\XKV@prefix
  \else
    \edef\XKV@header{\XKV@prefix\XKV@header @}%
  \fi
}
```

`\XKV@testopta` `\XKV@t@stopta` `\XKV@t@st@pta` `\XKV@@t@st@pta` Optional argument testing. Used for instance by `\setkeys`.

```
\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\XKV@t@st@pta#1}{\XKV@plfalse\XKV@t@st@pta#1}%
}
\def\XKV@t@st@pta#1{\@testopt{\XKV@@t@st@pta#1}{KV}}
\def\XKV@@t@st@pta#1[#2]#3{%
```

Set prefix.

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

Store families and sanitize commas.

```
  \XKV@checksanitizeb{#3}\XKV@fams
  \expandafter\XKV@sp@deflist\expandafter\XKV@fams\expandafter{\XKV@fams}%
  \@testopt#1{}%
}
```

\XKV@testoptb
\XKV@t@stoptb

Optional argument testing. Used for instance by `\define@key`.

```
\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@ifcmd
\XKV@@ifcmd

`{`⟨*tokens*⟩`}{`⟨*macro*⟩`}{`⟨*cmd*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}`
This macro checks whether the ⟨*tokens*⟩ contains the macro specification ⟨*macro*⟩. If so, the argument of 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}`.

```
\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
\XKV@g@tkeyname

Utility macros to retrieve the key name from `key=value`, `\savevalue{key}=value` or `\gsavevalue{key}=value`. `\ifXKV@rkv` will record whether this particular value should be saved or not. `\ifXKV@sg` will record whether this value should be saved globally or not.

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

Utility macro to check whether `key` or `\global{key}` has been specified in `\savekeys`.

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

`\define@key` Macro to define a key in a family. Original but modified keyval code. 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.

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

`\XKV@define@key` Workhorse for `\define@key`.

```
\def\XKV@define@key#1{%
```

Define the key macro.

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

`\XKV@d@fine@k@y` Defines a key macro and a default value macro.

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

`\define@boolkey` Defines a boolean key.

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

`\XKV@define@boolkey` Workhorse for `\define@boolkey`.

```
\def\XKV@define@boolkey#1{%
```

Create the conditional.

```
  \expandafter\newif\csname if\XKV@header#1\endcsname
```

Create the key function.

```
  \expandafter\edef\csname\XKV@header#1\endcsname##1%
    {\noexpand\XKV@setbool{\XKV@header#1}{##1}}%
  \@ifnextchar[{\XKV@d@fine@boolkey{#1}}{}%
}
```

`\XKV@d@fine@boolkey` Define the default value macro.

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

`\XKV@setbool` Set a boolean key.

```
\def\XKV@setbool#1#2{%
  \def\XKV@tempa{true}%
  \def\XKV@tempb{false}%
  \lowercase{\def\XKV@tempc{#2}}%
  \ifx\XKV@tempc\XKV@tempa\else
    \ifx\XKV@tempc\XKV@tempb\else
      \let\XKV@tempc\relax
    \fi
  \fi
  \ifx\XKV@tempc\relax
    \XKV@err{boolean can only be ‘true’ or ‘false’}%
  \else
```

```
    \csname#1\XKV@tempc\endcsname
  \fi
}
```

`\define@cmdkey` Defines a command key. This key will store its input in a macro for later use.

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

`\XKV@define@cmdkey` Workhorse for `\define@cmdkey`.

```
\def\XKV@define@cmdkey#1{%
```

Create the key function.

```
  \expandafter\edef\csname\XKV@header#1\endcsname##1{\noexpand\def
    \expandafter\noexpand\csname\XKV@header#1@cmd\endcsname{##1}}%
  \@ifnextchar[{\XKV@d@fine@cmdkey{#1}}{}%
}
```

`\XKV@d@fine@cmdkey` Defines the default value macro.

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

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

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

`\XKV@key@ifundefined` This macro is split in two parts so that `\XKV@p@x` can use only the main part of the macro.

```
\def\XKV@key@ifundefined[#1]#2{%
```

Set the prefix and save the key name and family names.

```
  \XKV@makepf{#1}%
  \XKV@sp@deflist\XKV@fams{#2}%
  \XKV@key@if@ndefined
}
```

`\XKV@key@if@ndefined` Workhorse for `\key@ifundefined`.

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

Loop over possible families.

```
  \XKV@whilist\XKV@tfam:=\XKV@fams\do\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` Macro that make a key produce a warning on use.

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

`\XKV@disable@keys` Workhorse for `\disable@keys` which redefines a key macro.

```
\def\XKV@disable@keys#1{%
  \XKV@checksanitizeb{#1}\XKV@tempa
  \@for\XKV@tempa:=\XKV@tempa\do{%
    \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` `\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` Executes 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` 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` `\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` Run the main macro for both head tail presets.

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

`\XKV@d@lpresetkeys` 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` `\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 ε-TeX is available (hence using `\ifx\csname`), but doesn't work when ε-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` 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` 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` 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 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 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 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. Use \XKV@testopta to handle optional arguments.

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

\XKV@setkeys Workhorse for \setkeys.

```
\def\XKV@setkeys[#1]#2{%
```

Macros to retrieve a list of keys from the user input.

```
  \def\XKV@tempa##1,{%
    \def\XKV@tempb{##1}%
    \ifx\XKV@tempb\@nnil\else
      \XKV@g@tkeyname##1=\@nil\XKV@tempb
      \XKV@addtolist@x\XKV@kna\XKV@tempb
      \expandafter\XKV@tempa
    \fi
  }%
  \XKV@checksanitizea{#2}\XKV@resb
  \let\XKV@kna\@empty
  \expandafter\XKV@tempa\XKV@resb,\@nil,%
```

Initialize the remaining keys.

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

Initialize the macro that should be executed.

```
  \let\XKV@exec\@empty
```

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}%
```

Execute all key macros.

```
  \XKV@exec
}
```

\XKV@usepresetkeys 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.

```
\def\XKV@usepresetkeys#1#2{%
  \XKV@for\XKV@tfam:=\XKV@fams\do{%
    \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@kna\ifx\XKV@kna\@empty\else,\fi#1}%
    }%
  }%
}
```

\XKV@s@tkeys This macro starts the loop over keys.

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

Define the list of key names which should be ignored.

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

Start the loop over keys.

```
  \XKV@s@tk@ys#1,\@nil,%
}
```

\XKV@s@tk@ys Workhorse for `\XKV@s@tkeys`.

```
\def\XKV@s@tk@ys#1,{%
  \def\XKV@tempa{#1}%
  \ifx\XKV@tempa\@nnil\else
    \XKV@knftrue
```

Split key and value.

```
    \XKV@split#1==\@nil
```

Check whether the key has been found.

```
    \ifXKV@knf
      \ifXKV@inpox
```

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

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

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

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

Pass the option through `\DeclareOptionX*`.

```
          \def\CurrentOption{#1}\XKV@doxs
        \fi
      \else
```

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

```
        \ifXKV@st
          \XKV@addtolist@n\XKV@rm{#1}%
        \else
          \XKV@err{`\XKV@tkey' undefined in families `\XKV@fams'}%
        \fi
      \fi
    \else
```

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

```
      \ifXKV@inpox\ifx\XKV@testclass\XKV@documentclass
        \XKV@useoption{#1}%
      \fi\fi
    \fi
    \expandafter\XKV@s@tk@ys
  \fi
}
```

`\XKV@split` Macro that splits keys and values.

```
\def\XKV@split#1=#2=#3\@nil{%
```

Remove spaces from key name and check for `\savevalue` and `\gsavevalue`.

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

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

```
    \@expandtwoargs\in@{,\XKV@tkey,}{,\XKV@kn,}%
    \ifin@\XKV@knffalse\else
      \KV@@sp@def\XKV@tempa{#2}%
```

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

```
      \XKV@ifundefined{XKV@\XKV@header save}{}{%
        \expandafter\XKV@testsavekey\csname XKV@\XKV@header save\endcsname\XKV@tkey
      }%
```

Save the value of a key.

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

Replace pointers by saved values.

```
      \expandafter\XKV@replacepointers\expandafter{\XKV@tempa}%
      \ifXKV@pl
```

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

```
        \XKV@for\XKV@tfam:=\XKV@fams\do{%
          \XKV@makehd\XKV@tfam
          \expandafter\XKV@setkey@infam\expandafter{\XKV@tempa}{#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@tfam:=\XKV@fams\do\ifXKV@knf\fi{%
          \XKV@makehd\XKV@tfam
          \expandafter\XKV@setkey@infam\expandafter{\XKV@tempa}{#3}%
        }%
      \fi
    \fi
  \fi
}
```

`\XKV@testsavekey` This macro checks whether the key in macro #2 appears in the save list in macro #1. 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
    \@for\XKV@resa:=#1\do{%
      \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`
`\XKV@r@placepointers` Replaces all pointers by their saved value. The result is stored in #4. We feed the replacement and the following tokens again to the macro to replace nested pointers. It stops when no pointers are found anymore.

```
\def\XKV@replacepointers#1{%
  \let\XKV@tempa\@empty
  \let\XKV@resa\@empty
  \XKV@r@placepointers#1\usevalue\@nil
}
\def\XKV@r@placepointers#1\usevalue#2{%
  \XKV@addtomacro@n\XKV@tempa{#1}%
  \ifx\@nil#2\relax\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@setkey@infam Sets a key in a family. Based on keyval code.

```
\def\XKV@setkey@infam#1#2{%
 \XKV@ifundefined{\XKV@header\XKV@tkey}{}{%
```

Check whether the key macro is defined.

```
  \XKV@knffalse
  \ifx\@empty#2\@empty
```

No value given, use default.

```
   \XKV@ifundefined{\XKV@header\XKV@tkey @default}{%
    \XKV@err{no value specified for key `\XKV@tkey'}%
   }{%
```

Execute key with the default value.

```
    \expandafter\expandafter\expandafter\XKV@default
     \csname\XKV@header\XKV@tkey @default\endcsname\@nil
   }%
  \else
```

Add key macro and its value to the execution macro.

```
   \XKV@addtomacro@o\XKV@exec{\csname\XKV@header\XKV@tkey\endcsname{#1}\relax}%
  \fi
 }%
}
```

\XKV@default 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 name of 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}%
```

Add the key macro with the (possibly changed) default value to the execution macro.

```
    \XKV@addtomacro@o\XKV@exec{\expandafter#1\expandafter{\XKV@tempa}\relax}%
  \else
```

Add the default value macro without any features to the execution macro.

```
    \expandafter\XKV@addtomacro@o\expandafter\XKV@exec\expandafter
      {\csname\XKV@header\XKV@tkey @default\endcsname\relax}%
  \fi
}
```

`\setrmkeys` 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@setrmkeys}
```

`\XKV@setrmkeys` 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
⟨/tex⟩
```

## 12.2 LaTeX program

Initialize the package.

```
%<*latex>
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesPackage{xkeyval}[2005/01/30 v2.0 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.tex \fi
\edef\XKVcatcodes{%
  \catcode`\noexpand\=\the\catcode`\=\relax
  \catcode`\noexpand\,\the\catcode`\,\relax
  \catcode`\noexpand\:\the\catcode`\:\relax
  \let\noexpand\XKVcatcodes\relax
}
\catcode`\=12\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}
```

At loading, retrieve document class, copy `\@classoptionslist` to `\XKV@classoptionslist` and filter `key=value` pairs from the original.

```
\ifx\XKV@documentclass\@undefined
```

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\XKV@tempa:=\@filelist\do\ifx\XKV@documentclass\@undefined\fi{%
    \filename@parse\XKV@tempa
    \ifx\filename@ext\@clsextension
      \edef\XKV@documentclass{\filename@base.\filename@ext}%
    \fi
  }
  \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@tempb#1,\@nil,%
    }
    \def\XKV@tempb#1,{%
      \def\XKV@tempa{#1}%
      \ifx\XKV@tempa\@nnil\else
        \in@{=}{#1}%
      \ifin@\else\XKV@addtolist@n\@classoptionslist{#1}\fi
      \expandafter\XKV@tempb
    \fi
  }
  \expandafter\XKV@tempa\expandafter{\@classoptionslist}
 \fi
\fi
```

`\XKV@testoptc` `\XKV@t@stoptc` `\XKV@t@st@ptc` `\XKV@@t@st@ptc`

Macros for `\ExecuteOptionsX` and `\ProcessOptionsX` for testing for optional arguments and inserting default values.

```
\def\XKV@testoptc#1{%
  \XKV@ifstar{\XKV@sttrue\XKV@t@stoptc#1}{\XKV@stfalse\XKV@t@stoptc#1}%
}
\def\XKV@t@stoptc#1{\@testopt{\XKV@t@st@ptc#1}{KV}}
\def\XKV@t@st@ptc#1[#2]{%
  \XKV@makepf{#2}%
  \@ifnextchar<{\XKV@@t@st@ptc#1}{\XKV@@t@st@ptc#1<\@currname.\@currext>}%
}
\def\XKV@@t@st@ptc#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 an 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` `\XKV@@d@x` `\XKV@@@d@x`

Insert default prefix and family name (which is the filename of the class or package) and add empty default value if none present. Execute `\define@key`.

```
\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`

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@stoptc` 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@stoptc\XKV@setkeys}
```

`\ProcessOptionsX`

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@testoptc` to handle optional arguments.

```
\def\ProcessOptionsX{\XKV@stfalse\XKV@plfalse\XKV@testoptc\XKV@pox}
```

\XKV@pox 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` 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
⟨/latex⟩
```

## 12.3 LaTeX kernel patch

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].

```
%<*ltxpatch>
%%
%% Based on latex.ltx.
%%
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesPackage{xkvltxp}[2004/12/13 v1.2 LaTeX2e kernel patch (HA)]
\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
}
\def\@fileswith@pti@ns#1[#2]#3[#4]{%
  \ifx#1\@clsextension
    \ifx\@classoptionslist\relax
      \@temptokena{#2}%
      \xdef\@classoptionslist{\the\@temptokena}%
      \def\reserved@a{%
        \@onefilewithoptions#3[#2][#4]#1%
        \@documentclasshook}%
    \else
      \def\reserved@a{%
        \@onefilewithoptions#3[#2][#4]#1}%
    \fi
  \else
    \@temptokena{#2}%
    \def\reserved@b##1,{%
      \ifx\@nil##1\relax\else
        \ifx\relax##1\relax\else
         \noexpand\@onefilewithoptions##1[\the\@temptokena][#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
⟨/ltxpatch⟩
```

## 12.4 keyval primitives

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.

```
%<*keyval>
%%
%% Based on keyval.sty.
%%
\def\XKV@tempa#1{%
\def\KV@@sp@def##1##2{%
  \futurelet\XKV@resa\KV@@sp@d##2\@nil\@nil#1\@nil\relax##1}%
\def\KV@@sp@d{%
  \ifx\XKV@resa\@sptoken
    \expandafter\KV@@sp@b
  \else
    \expandafter\KV@@sp@b\expandafter#1%
  \fi}%
\def\KV@@sp@b#1##1 \@nil{\KV@@sp@c##1}%
  }
\XKV@tempa{ }
\def\KV@@sp@c#1\@nil#2\relax#3{\XKV@toks{#1}\edef#3{\the\XKV@toks}}
\def\KV@do#1,{%
 \ifx\relax#1\@empty\else
  \KV@split#1==\relax
  \expandafter\KV@do\fi}
\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}
⟨/keyval⟩
```

## 12.5 TEX header

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`.

```
%<*header>
%%
%% Taken from latex.ltx.
%%
\message{2005/01/02 v1.0 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}]}}
\def\@fornoop#1\@@#2#3{}
\long\def\@for#1:=#2\do#3{%
  \expandafter\def\expandafter\@fortmp\expandafter{#2}%
  \ifx\@fortmp\@empty \else
    \expandafter\@forloop#2,\@nil,\@nil\@@#1{#3}\fi}
\long\def\@forloop#1,#2,#3\@@#4#5{\def#4{#1}\ifx #4\@nnil \else
      #5\def#4{#2}\ifx #4\@nnil \else#5\@iforloop #3\@@#4{#5}\fi\fi}
\long\def\@iforloop#1,#2\@@#3#4{\def#3{#1}\ifx #3\@nnil
      \expandafter\@fornoop \else
     #4\relax\expandafter\@iforloop\fi#2\@@#3{#4}}
\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\zap@space#1 #2{%
  #1%
  \ifx#2\@empty\else\expandafter\zap@space\fi
  #2}
\def\strip@prefix#1>{}
\def \@onelevel@sanitize #1{%
  \edef #1{\expandafter\strip@prefix
           \meaning #1}%
}
⟨/header⟩
```

## References

[1] Hendri Adriaens. pst-xkey package, v1.3, 2005/01/16. `CTAN:/macros/latex/contrib/xkeyval`.

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

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

[4] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, and Chris Rowley. *The LaTeX Companion, Second Edition.* Addison-Wesley, 2004.

[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 Josselin Noirel, Till Tantau, Herbert Voß and Carsten Heinz for help and suggestions. Thanks go to Donald Arseneau for contributing the `\@ifnextcharacter` macro. Special thanks go to Uwe Kern for his ideas for improving the functionality of this package, a lot of useful comments on the package and the documentation and for contributing the `\@selective@sanitize` macro.

## Version history

**v1.0** *(2004/04/29)*
General: Initial release . . . . . . . . 1

**v1.1** *(2004/04/30)*
`\XKV@@@d@x`: Made to insert an empty default value if none present for `\DeclareOptionX` 43

**v1.2** *(2004/05/08)*
General: Change to `\DeclareOptionXf`; macro is now replaced . . . . . . 1

**v1.3** *(2004/05/09)*
General: Moved the options section to the end of the package to allow it to use xkeyval option macros . . . . . . . . . . . . . . . . 45
Revision of documentation . . . 1

**v1.4** *(2004/08/24)*
`\@ifnextcharacter`: Added robust next character check . . 23
General: Added keyval primitives 47
Added + option to macros . . . 7
Added pointer syntax . . . . . . . 8
Added prefix options to macros . . . . . . . . . . . . . . . . . . . 3, 6, 15
Changed package options . . . . 20
Fixed small bug in class option filtering . . . . . . . . . . . . . . . 42
Made package TeX compatible 1
Renamed macros to keyval names . . . . . . . . . . . . . . . . . 1
`\define@key`: Added optional check . . . . . . . . . . . . . . . . . 32
`\ProcessOptionsX`: Fixed macro for `\LoadClass` case . . . . . . 43
`\XKV@split`: Made macro more efficient . . . . . . . . . . . . . . . . . 38

**v1.5** *(2004/09/27)*
General: Added LaTeX kernel patch . . . . . . . . . . . . . . . . . 45
Added pst-xkey . . . . . . . . . . . 1
Added boolean keys . . . . . . . . 4
Added preset system . . . . . . . 11
Corrected some minor bugs . . 1
Made macros avoid expansions of options . . . . . . . . . . . . . . 42
Removed `\ifrecordkeyvals` . . 8
Revised documentation . . . . . 1
`\XKV@for`: Added . . . . . . . . . . . . 24
`\XKV@whilist`: Changed behavior 24

**v1.6** *(2004/10/05)*
General: Changed loading preventions . . . . . . . . . . . . . . . . . . 1
Corrected typos . . . . . . . . . . . 1
`\XKV@wh@list`: Added missing % 24

**v1.7** *(2004/11/25)*
General: Added `\disable@keys` . 5
Added command keys . . . . . . 5
Added system to control the scope of definitions to various macros . . . . . . . . . . . . . . . . . 1
Changed LaTeX patch from `.tex` to `.sty` . . . . . . . . . . . . . . . . 1
Changed name of `\XKV@ifku` . 5
Changed pointer mechanism . . 8
Changed value saving mechanism . . . . . . . . . . . . . . . . . . . 8
Improved coding . . . . . . . . . . 1
Package uses $\varepsilon$-TeX if available 1
Removed optional key checking from define macros . . . . . . . . 3
Updated license information . . 1
`\gpresetkeys`: Made incremental 34
`\XKV@makepf`: Made macro protect internal macros . . . . . . . . . . 30

**v1.8** *(2004/12/13)*
`\@s@l@ctive@sanitize`: Bug fixed, added level control . . . 26
General: Fixed inconsistency of treatment of spaces in key and family names, all left untouched now . . . . . . . . . . . . 1
Made package robust for catcode changes of equality character . . . . . . . . . . . . . . . . . 1
`\XKV@addtolist@n`: Changed to respect groups . . . . . . . . . . 25
`\XKV@merge`: Simplified . . . . . . . 28

**v1.8a** *(2004/12/20)*
`\XKV@setkeys`: Solved small bug . 36
`\XKV@sp@deflist`: Modified to respect the empty family . . . . 29

**v1.8b** *(2004/12/22)*
`\XKV@ch@cksanitize`: Fixed bug for key value `\@empty` . . . . . 27

**v1.8c** *(2005/01/01)*
General: Simplified and improved some code . . . . . . . . . . . . . 1

**v1.8d** *(2005/01/02)*
General: Renamed `xkeyval.def` to `xkvtxhdr.tex` . . . . . . . . . . . . 1
Restructured package to indicate general tools and their implementation . . . . . . . . . . . 1

`\XKV@ifundefined`: Simplified definition by using `\XKV@afterelsefi` . . . . . . . 23

`\XKV@r@placepointers`: Added `\XKV@afterfi` to avoid capacity problems when many pointers present . . . . . . . . . . . . 39

**v1.9** *(2005/01/16)*

General: Added `\delpresetkeys` 1

Added `\delsavekeys` . . . . . . . 1

Updated license information . . 1

`\XKV@@d@x`: Added extension to default family . . . . . . . . . . . . 43

`\XKV@r@placepointers`: Added back linking test . . . . . . . . . 39

`\XKV@t@st@ptc`: Added extension to default family . . . . . . . . . 43

`\XKV@wh@list`: Made running command contain last used list entry at exit . . . . . . . . . . . . 24

**v2.0** *(2005/01/30)*

General: Made `\setkeys` nestable 1

`\XKV@addtolist@n`: Simplified . . 25

`\XKV@addtolist@o`: Simplified . . 25

`\XKV@default`: Repaired adding extra braces when executing default value . . . . . . . . . . . . 40

`\XKV@ifundefined`: Made none $\varepsilon$-TeX version not leave `\relax` 23

`\XKV@r@placepointers`: Simplified . . . . . . . . . . . . . . . . . . 39

## 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.