The skeyval Package*

Version 0.5

Ahmed Musa ✉
Preston, Lancashire, UK

4th March 2010

Abstract

This package supplements the xkeyval package, hence the "s" in "skeyval." It introduces toggle keys and complementary (boolean and toggle) keys. It also provides mechanisms for reserving, unreserving, suspending, restoring, and removing keys. Furthermore, it introduces a set of commands for key definition which bar the developer or user from inadvertently redefining existing keys of the same family and prefix. Commands are provided for checking the statuses of keys across multiple key prefixes and families. Also, the package provides a scheme for defining multiple keys of different genres using only one command, thereby making it possible to considerably economize on tokens when defining keys. Some other general-purpose developer macros are provided within the package.

*The package was formerly called the keyreader package until version 0.5. The keyreader package is obsolete and no longer supported.

The skeyval package is available at http://www.ctan.org/tex-archive/macros/latex/contrib/skeyval/

Contents

1 Motivation

Toggle switches or booleans were introduced by the etoolbox package and have proved very useful mainly for two reasons: unlike the legacy TEX switches which require three commands per switch, toggles require only one command per switch, and toggles occupy their own separate namespace, thereby avoiding clashes with other macros. So we can effectively have both the following sets in the same file:

Example

```
\newif\ifmyboolean    ->  3 separate commands:
                          \ifmyboolean \mybooleantrue
                          \mybooleanfalse

\newtoggle{myboolean} ->  only 1 command and no clash with
                          commands in other namespaces.
```

However, the xkeyval package can't be used to define and set toggle keys. The present package fills this gap, by providing facilities for defining and setting toggle keys. The work relies on some of the macros from the xkeyval package.

Secondly, the xkeyval package can't be used to define and set complementary keys, which can be handy in the case of boolean and toggle keys. The present package introduces this concept and additionally permits the submission of individual/different custom key macros to complementary boolean and toggle keys.

The third motivation for this package relates to economy of tokens in style files. The xkeyval package provides \define@cmdkeys and \define@boolkeys for defining and setting multiple command keys and boolean keys, but in each category the keys must have the same default value and no key macro/function. This package seeks to lift these restrictions, so that multiple keys of all categories (ordinary keys, command keys, boolean keys, tog keys, and choice keys) can be defined in one go (using only one command) and those keys can have different default values and functions. This greatly minimizes tokens, as hundreds of keys can, in principle, be issued simultaneously by one command.

Fourthly, macros are introduced for defining all key types without the fear of inadvertently redefining existing keys in the same family and with the same key prefix. This is similar to the \newcommand concept in LaTeX.

The package also provides facilities for disabling, suspending, restoring and removing keys across multiple families of keys.

The new macros can be used together with the machinery from the xkeyval package for efficient and versatile key management.

2 Loading the skeyval package

The package can be loaded in style and class files by

Example

```
\RequirePackage[options]{skeyval}
```

and in document files via

Example

```
\usepackage[options]{skeyval}
```

where the options and their default values are

Macro

```
keyparser=;, macroprefix=mp@, keyprefix=KV, keyfamily=fam,
xchoicelist=false.
```

The ⟨keyparser⟩ is the separator between the keys in the key list to be defined in one go (see examples in Section 6.3). The ⟨macroprefix⟩, ⟨keyprefix⟩, and ⟨keyfamily⟩ are, respectively, the macro prefix, key prefix and key family for all

the keys to be defined upon the declaration of these options. All these options can be set or changed dynamically by using the \skvoptions macro:

Macro

```
\skvoptions{keyparser=;, macroprefix=mp@, keyprefix=KV,
  keyfamily=fam, xchoicelist=false}.
```

If, as unlikely as it may seem, a clash arises between package and/or user macros as a result of the use of the defaults for ⟨macroprefix⟩, ⟨keyprefix⟩ and ⟨keyfamily⟩, then the user will have to make his own choices for these defaults so as to avoid clashes.

3 Complementary boolean keys

The syntax for creating complementary boolean keys is

Macro

```
\define@compboolkeys[⟨keyprefix⟩]{⟨family⟩}[⟨macroprefix⟩]
  {⟨primary boolean⟩}[⟨default value for primary boolean⟩]
  {⟨secondary boolean⟩}{⟨func for primary boolean⟩}
  {⟨func for secondary boolean⟩}.
```

This command is robust and can be used in expansion contexts, but expandable commands must be protected. When the user doesn't supply the ⟨keyprefix⟩ and/or ⟨macroprefix⟩, the package will use ⟨KV⟩ and ⟨mp@⟩, respectively. When one complementary boolean key (primary or secondary) is true, the other is automatically set false; and vice versa: when one complementary boolean key (primary or secondary) is false, the other is automatically set true. Infinite loops, which are possible in back-linked key settings, are avoided in the skeyval package. The tools of the xkeyval package, such as \setkeys, \presetkeys, \savekeys, \savevalue, \usevalue, etc., are all applicable to complementary boolean keys.

As an example, we define below two complementary boolean keys ⟨draft⟩ and ⟨final⟩ with different key macros:

Example

```
\define@compboolkeys[KV]{fam}[mp@]{draft}[true]{final}%
  {%
    \def\gobble##1{}%
  }{%
    \def\notgobble##1{##1}%
  }.
```

The key prefix (default ⟨KV⟩), macro prefix (default ⟨mp@⟩), and key macros (no default) can be empty:

Example

```
23 \define@compboolkeys{fam}{draft}[true]{final}{}{}.
```

The defined complementary keys ⟨draft⟩ and ⟨final⟩ can now be set as follows:

Example

```
24 \setkeys[KV]{fam}{draft=true}

26 \setkeys[KV]{fam}{final=true}
```

The second statement above reverses the boolean ⟨draft⟩ to ⟨false⟩, which had been set in the first statement to ⟨true⟩. There is no apparent meaning to the following:

Example

```
27 \setkeys[KV]{fam}{draft=true,final=true}.
```

Most applications of the xkeyval package do indeed use key and macro prefixes; so it presumably makes sense here to assume that all uses of the present package will somehow involve key and macro prefixes.

4 Toggle switches and keys

4.1 Toggle switches

The following toggle switches are defined in the skeyval package. They largely mimic those in the etoolbox package, except for the commands \deftog and \requiretog. There is no fear that the commands in this package will interfere with those from the etoolbox package, since the control sequence names used in the two packages are different.

All the commands in this section are robust and can be used in expansion contexts.

Macro

```
28 \deftog{⟨toggle⟩}
```

This defines a new ⟨toggle⟩ whether or not ⟨toggle⟩ is already defined. If ⟨toggle⟩ is already defined, a warning message is logged in the transcript file and the new definition is effected.

Macro

```
29 \newtog{⟨toggle⟩}
```

This defines a new ⟨toggle⟩ if ⟨toggle⟩ is not already defined; otherwise the package issues a fatal error.

Macro

30 `\providetog{⟨toggle⟩}`

This defines a new ⟨toggle⟩ if ⟨toggle⟩ is not already defined. If ⟨toggle⟩ is already defined, the command does nothing.

Macro

31 `\requiretog{⟨toggle⟩}`

`\requiretog` takes arguments like `\newtog` and behaves like `\providetog` with the difference: if the toggle is already defined, the command `\requiretog` calls LaTeX's `\CheckCommand` to make sure that the new and existing definitions are identical, whereas `\providetog` assumes that if the toggle is already defined, the existing definition should persist. `\requiretog` assures that a toggle will have the given definition, but `\requiretog` also warns the user if there was a previous and different existing definition. For example, if the toggle ⟨toga⟩ is currently ⟨true⟩, then since all new toggles start out as ⟨false⟩, a call `\requiretog{toga}` will issue a warning in the log file that the new and old definitions of ⟨toga⟩ don't agree and the new definition, therefore, can't go ahead.

The `skeyval` package also provides the command `\requirecmd`, which has the same logic as `\requiretog` but can be used for general LaTeX commands, including those with optional arguments (see Section 12).

Macro

32 `\settog{⟨toggle⟩}{⟨true | false⟩}`

This command sets ⟨toggle⟩ to ⟨value⟩, where ⟨value⟩ may be either ⟨true⟩ or ⟨false⟩. This statement will issue an error if ⟨toggle⟩ wasn't previously defined.

Macro

33 `\togtrue{⟨toggle⟩}`

This sets ⟨toggle⟩ to ⟨true⟩. It will issue an error if ⟨toggle⟩ wasn't previously defined.

Macro

34 `\togfalse{⟨toggle⟩}`

This sets ⟨toggle⟩ to ⟨false⟩. It will issue an error if ⟨toggle⟩ wasn't previously defined.

Macro

```
35 \iftog{⟨toggle⟩}{⟨true⟩}{⟨false⟩}
```

This yields the ⟨true⟩ statement if the boolean ⟨toggle⟩ is currently ⟨true⟩, and ⟨false⟩ otherwise. It will issue an error if ⟨toggle⟩ wasn't previously defined.

Macro

```
36 \ifnottog{⟨toggle⟩}{⟨not true⟩}{⟨not false⟩}
```

This behaves like \iftog but the logic of the test is reversed. It will issue an error if ⟨toggle⟩ wasn't previously defined.

4.2 Toggle keys

The syntax for defining toggle keys is exactly like those for boolean keys in the xkeyval package. This allows all the machinery of the xkeyval package (including \setkeys, \presetkeys, \savekeys, \savevalue, \usevalue, etc) to be applicable to toggle keys.

As mentioned earlier, toggles have their own separate namespace. However, the \setkeys command (and friends) of the xkeyval package is unaware of this. This can cause problems when the user uses the same name for boolean and toggle keys (or indeed any key type) in the same family and with the same key prefix, believing rightly that toggle keys have their own separate namespace. If this is a source of significant concern to any user, he will be well advised to instead use the commands \newboolkey, \newboolkeys, \newtogkey, \newtogkeys, etc., of Section 9. In those commands a mechanism is included to bar keys from having the same name as other keys in the same family and with the same prefix. Toggle keys can still share the same names with keys across families and key prefixes. Since it is not always certain which of the keys the user may want to first define (before its definition is possibly repeated), the fear of interference has necessitated new syntaxes for defining all key types, which completely avoid interference (see Section 9).

The user interfaces for defining toggle keys are

Macro

```
37 \define@togkey[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨key⟩}[⟨default⟩]%
38   {⟨function⟩}

40 \define@togkey+[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨key⟩}[⟨default⟩]%
41   {⟨function1⟩}{⟨function2⟩}
```

If the macro prefix ⟨mp⟩ is not specified, these create a toggle of the form ⟨prefix⟩@⟨family⟩@⟨key⟩ using \deftog (which initializes the toggle switch to

⟨false⟩) and a key macro of the form \⟨prefix⟩@⟨family⟩@⟨key⟩ which first checks the validity of the user input. If the value is valid, it uses it to set the toggle and then executes ⟨function⟩. If the user input wasn't valid (i.e., neither true nor false), then the toggle will not be set and the package will generate a fatal error to this effect.

If ⟨mp⟩ is specified, then the key definition process will create a toggle of the form ⟨mp⟩⟨key⟩ and a key macro of the form \⟨mp⟩⟨key⟩. The value ⟨default⟩ will be used by the key macro when the user sets the key without a value.

If the plus (+) version of the macro is used, the user can specify two key macros ⟨function1⟩ and ⟨function2⟩. If user input is valid, the macro will set the toggle and executes ⟨function1⟩; otherwise, it will not set the boolean but will execute ⟨function2⟩.

As an example, consider the following (adapted from the xkeyval package to suit toggle keys):

Example

```
\define@togkey{fam}[my@]{frame}{%
  \iftog{my@frame}{%
    \PackageInfo{mypack}{Turning frames on}%
  }{%
    \PackageInfo{mypack}{Turning frames off}%
  }%
}

\define@togkey+{fam}{shadow}{%
  \iftog{KV@fam@shadow}{%
    \PackageInfo{mypack}{Turning shadows on}%
  }{%
    \PackageInfo{mypack}{Turning shadows off}%
  }%
}{%
    \PackageWarning{mypack}{Erroneous input `#1' ignored}%
}
```

The first example creates the toggle ⟨my@frame⟩ and defines the key macro \KV@fam@frame to set the boolean (if the input is valid). The second key intimates the user of changed settings, or produces a warning when input was incorrect.

It is also possible to define multiple toggle keys with a single command:

Macro

```
\define@togkeys[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨keys⟩}[⟨default⟩]
```

This creates a toggle key for every entry in the comma-separated list ⟨keys⟩. As is the case with the commands \define@cmdkeys and \define@boolkeys from the xkeyval package, the individual keys in this case can't have a custom function. See section 6 for how to define multiple keys with custom functions.

As an example of defining multiple toggle keys, consider

Example

```
60 \define@togkeys{fam}[my@]{toga,togb,togc}
```

This is an abbreviation for

Example

```
61 \define@togkey{fam}[my@]{toga}{}
62 \define@togkey{fam}[my@]{togb}{}
63 \define@togkey{fam}[my@]{togc}{}
```

Now we can do

Example

```
64 \define@togkey{fam}[my@]{book}{%
65   \iftog{my@book}{\setkeys[KV]{fam}{togc=true}}{}%
66 }
67 \setkeys[KV]{fam}{book=true}
```

Toggle keys can be set in the same way that other key types are set.

The status of toggles can be examined by doing

Example

```
68 \show\SKV@toggle@⟨mp⟩⟨key⟩
```

when the ⟨mp⟩ is present. When the user has specified no ⟨mp⟩ in defining the key, he has to issue

Example

```
69 \show\SKV@toggle@⟨prefix⟩@⟨family⟩@⟨key⟩.
```

5 Complementary toggle keys

Similar to complementary boolean keys of Section 3, the skeyval package introduces a facility for creating complementary toggle keys. The syntax for this is similar to that for defining complementary toggle keys:

Macro

```
\define@comptogkeys[⟨keyprefix⟩]{⟨family⟩}[⟨macroprefix⟩]
  {⟨primary toggle⟩}[⟨default value for primary toggle⟩]
  {⟨secondary toggle⟩}{⟨func for primary toggle⟩}
  {⟨func for secondary toggle⟩}.
```

This command is robust and can be used in expansion contexts, but non-robust commands have to be protected. When the user doesn't supply the ⟨keyprefix⟩ and/or ⟨macroprefix⟩, the package will use ⟨KV⟩ and ⟨mp@⟩, respectively. When one toggle key (primary or secondary) is true, the other is automatically set false; and vice versa: when one toggle key (primary or secondary) is false, the other is automatically set true.

As an example, we define below two complementary toggle keys ⟨xdraft⟩ and ⟨xfinal⟩ with different key macros:

Example

```
\define@comptogkeys[KV]{fam}[mp@]{xdraft}[true]{xfinal}%
  {%
    \def\gobble##1{}%
  }{%
    \def\notgobble##1{##1}%
  }.
```

The key prefix (whose default is ⟨KV⟩), macro prefix (whose default is ⟨mp@⟩), and key macros (no default) can be empty:

Example

```
\define@comptogkeys{fam}{xdraft}[true]{xfinal}{}{}.
```

The defined complementary toggle keys ⟨xdraft⟩ and ⟨xfinal⟩ can now be set as follows:

Example

```
\setkeys[KV]{fam}{xdraft=true}

\setkeys[KV]{fam}{xfinal=true}
```

The second statement above reverses the toggle ⟨xdraft⟩ to ⟨false⟩, which had been set in the first statement to ⟨true⟩.

Toggle keys may easily be confused with the conventional boolean keys, at the time of definition and setting. It is therefore always safer to use the syntaxes in Section 9 for defining keys; they avoid interference between new and existing keys.

Note If we were to use the key names `draft` and `final` as toggle keys above, instead of `xdraft` and `xfinal`, there would have been a clash with the keys `draft` and `final` defined as (complementary) boolean keys in Section 3— because they share the same family ⟨`fam`⟩ and prefix ⟨`KV`⟩. The names `draft` and `final` can be used as toggles only if the family ⟨`fam`⟩ or prefix ⟨`KV`⟩ is changed.

6 Defining multiple keys of all genres by one command

The interface for defining multiple keys of all kinds in one go is the command `\define@keylist`, whose syntax is

Macro

```
\define@keylist{⟨key type/id⟩, ⟨key⟩, ⟨key default value⟩,
  ⟨key macro/function⟩; ⟨another set of key specifiers⟩; etc}
```

There are five key types: 1 (ordinary key), 2 (command key), 3 (boolean key), 4 (toggle key), and 5 (choice key). The key and its attributes are separated by commas; they constitute one `object`. The objects are separated by the ⟨`keyparser`⟩, which is the semicolon in the above example.

If the key list is available in a macro, say,

Example

```
\def\keylist{⟨key type/id⟩, ⟨key⟩, ⟨key default value⟩,
  ⟨key macro/function⟩; ⟨another set of key specifiers⟩; etc},
```

then the keys can be defined by the starred form of `\define@keylist`:

Example

```
\define@keylist*\keylist.
```

`\define@keylist*` takes a macro as argument, while `\define@keylist` accepts a key list.

The `\define@keylist` macro uses the following commands in the background:

Example

```
\define@key, \define@cmdkey, \define@boolkey,
\define@choicekey, \define@togkey.
```

Therefore, it assumes that it is safe to redefine a previously defined key. If this assumption is unwarranted, then the user should consider using the machinery

of Section 9[1].

6.1 Choice key values

The \ChoiceKeyValues macro is needed for choice keys; it lists the alternate admissible values for a choice key and thus can't be empty when a choice key is being defined. Its syntax is

Macro

```
\ChoiceKeyValues{⟨key⟩}{⟨list⟩},
```

where ⟨list⟩ is a comma-separated list of admissible key values. To further save tokens, the macro \ChoiceKeyValues may be abbreviated by \CKVS. It has to be defined each time a choice key is being defined. For example, if we want to define two choice keys align and election, then before the call to \define@keylist, we have to set

Example

```
\CKVS{align}{center,right,left,justified}
\CKVS{election}{state,federal,congress,senate}.
```

It doesn't matter which choice key first gets a \CKVS. The prevailing key prefix and key family are used internally by \ChoiceKeyValues to build distinct alternate values lists for choice keys.Unless the key family changes, you can't set two \ChoiceKeyValues for the same choice key. This will be possible only if the package option xchoicelist (meaning "allow overwriting of choice list") has been set ⟨true⟩, either through \documentclass, \usepackage, or \skvoptions. Therefore, any number of choice keys are allowed to appear in one \define@keylist or \define@keylist* statement if their lists of alternate/admissible values have been set by \CKVS.

As mentioned earlier, the key family and other package options can be changed dynamically via

Example

```
\skvoptions{keyparser=value,macroprefix=value,keyprefix=value,
  keyfamily=value,xchoicelist=value}.
```

In line with the philosophy of the xkeyval package, all the choice keys to be defined using the skeyval package require \ChoiceKeyValues: choice keys, by definition, have pre-ordained or acceptable values.

[1] In fact, I now often use the machinery of Section 9 to safely define new keys without the fear of inadvertently redefining an existing key within the same family and with the same key prefix.

6.2 Some internals

The internal equivalent of \ChoiceKeyValues (the choice key list of alternative values) is the macro \⟨keyprefix⟩@⟨keyfamily⟩@⟨key⟩@⟨altlist⟩. For example, for a key align in the family fam, and with prefix KV, the internal of \CKVS is \KV@fam@align@altlist.

For all keys in a family, the internal of the key macro (provided at key definition time) can be accessed via the macros

Macro

```
96 \⟨keyprefix⟩@⟨keyfamily⟩@⟨key⟩@⟨func⟩.
```

These internals are available for only the keys defined via \define@keylist or \newkeylist!

It should be noted that the xkeyval package will save and provide a key value when the value is saved (using the pointers \savevalue or \gsavevalue) at the time the key is set, or when the key is included in the \savekeys (or \gsavekeys) list. In that case, the saved value will be available in

Example

```
97 \XKV@⟨keyprefix⟩@⟨keyfamily⟩@⟨key⟩@⟨value⟩,
```

where the prefix XKV@ is usually added. The saved value can also be accessed via \usevalue{⟨key⟩} but only within \setkeys command. Since the skeyval package uses the machinery of the \xkeyval in the background, all these pointers can still be utilized for the keys defined via all the new key definition mechanisms introduced by the skeyval package (e.g., the commands \define@togkey, \define@keylist, \newkeylist, etc.).

The macro \⟨keyprefix⟩@⟨keyfamily⟩@⟨key⟩@⟨func⟩ is undefined if ⟨key⟩ has not been defined or if it has been removed; and the macro

Example

```
98 \XKV@⟨keyprefix⟩@⟨keyfamily⟩@⟨key⟩@⟨value⟩
```

is undefined whenever ⟨key⟩ has no value specified, or has not been set. So it is advisable to always test for the existence of these macros before they are used.

6.3 Some examples

In this section we provide a glimpse of the potential applications of the tools provided by the skeyval package in the context of defining multiple keys by one command.

Suppose that the key family and other attributes have been set as

Example

```
\skvoptions{keyparser=;,macroprefix=mp@,keyprefix=KV,
  keyfamily=fam,xchoicelist=false}.
```

Further, suppose we wish to define a set of keys ⟨color,angle,scale,align⟩. The keys color, angle and scale will de defined as command keys, while the key align will be defined as a choice key. Assume that the align key can only assume one of the values ⟨center | right | left | justified⟩, where the first three values would further imply \centering, \flushright, and \flushleft, respectively. Moreover, we assume that the key scale will be associated with a macro called \mydo, which depends on a previously defined macro \do. Together with align, we would also like to define another choice key: weather. The keys color and angle aren't associated with macros. Then we can do:

Example

```
\CKVS{align}{center,right,left,justified}
\CKVS{weather}{sunny,cloudy,lightrain,heavyrain,snow,
  sleet,windy,\someweather}
  %  We assume that \someweather is defined
  %  somewhere and holds an admissible value
  %  for the key ``weather'' at any level.
\def\f@align{%
  \ifcase\nr\relax
    \def\mp@align{\centering}%
  \or
    \def\mp@align{\flushright}%
  \or
    \def\mp@align{\flushleft}%
  \or
    \let\mp@align\relax
  \fi
}

\define@keylist{2,color,gray!25,;2,angle,45,;
  2,scale,1,\def\mydo##1{\do ##1};5,align,center,\f@align;
  \stopread;3,mybool,true,;
  5,weather,sunny,\protected@edef\VWeather{\val}}.
```

The \nr and \val macros are bin parameters for choice keys, as defined by the xkeyval package. \val contains the user input for the current key and \nr contains the numeral corresponding to the user input in the \CKVS list, starting from 0 (zero). For example, in the \CKVS{align} list, the \nr values are center (0), right (1), left (2), and justified (3). These parameters thus refresh with the choice key and its user-supplied value.

Instead of defining the macro \f@align before hand, we can submit its replacement text directly to the macro \define@keylist, but, because \f@align contains a conditional, some care is needed in doing so (see section 8). Once the key align has been defined, the macro \f@align can be reused—perhaps to define other keys—even before the key align is set. This is because it isn't \f@align that is used in defining the key align but its internal counterpart (i.e., a prefix and family-dependent internal of \f@align, which is \KV@fam@align@func). In this way, the user can economize on tokens. The same applies to all the macros that may be used in defining keys via \define@keylist.

Note the \stopread command inserted above. Because of it, the key mybool will not be read and defined; the rest (i.e., color, angle, scale and align) will be read and defined. All the entries for mybool and weather will instead be saved in the macro \SKV@remainder, possibly for some other uses.

Hundreds of keys can be defined efficiently in this way, using very few tokens.

As another example, we consider the following keys:

Example

```
\CKVS{align}{center,right,left,justified}
\CKVS{election}{state,federal,congress,senate}
% \CKVS needs to be defined only once for each key in a family.

\define@keylist{%
  3,boolvar,true,;1,paperheight,\paperheight,;
  1,paperwidth,\paperwidth,\f@paperwidth;
  2,textheight,\textheight,\f@textheight;
  2,textwidth,\textwidth,\f@textwidth;
  1,evensidemargin,\evensidemargin,;
  5,align,center,\f@align;
  5,election,congress,;
  2,testdim,2cm,\long\def\f@testdim##1{A test dimension ##1
    \par\bigskip}%
  %  Note the number of parameter characters
  %  in the definition of \f@testdim.
}
```

which have the following trivial key macros:

Example

```
\def\f@textwidth{\AtBeginDocument{\wlog{‘textwidth’ %
  is \mp@textwidth}}}

\def\f@textheight{%
  \ifx\@empty\mp@textheight
    \wlog{‘textheight’ value empty}%
  \else
```

```
    \wlog{‘textheight’ value not empty}%
  \fi
}

\def\f@paperwidth{\wlog{‘paperwidth’ was defined as %
  ordinary key.}}
\newcommand\f@align{%
  \ifcase\nr\relax
    \def\mp@align{\centering}%
  \or
    \def\mp@align{\flushright}%
  \or
    \def\mp@align{\flushleft}%
  \or
    \let\mp@align\relax
  \fi
}
```

Again, once the keys have been defined, these macros can be reused.

The same set of keys can be defined via the starred form of `\define@keylist`:

Example

```
\def\keylist{%
  3,boolvar,true,;1,paperheight,\paperheight,;
  1,paperwidth,\paperwidth,\f@paperwidth;
  2,textheight,\textheight,\f@textheight;
  2,textwidth,\textwidth,\f@textwidth;
  1,evensidemargin,\evensidemargin,;
  4,mytoggle,true,\let\settoggle\settog;
  5,align,center,\f@align;
  5,election,congress,;
  2,testdim,2cm,\long\def\f@testdim##1%
    {Do something with ##1}%
}
\define@keylist*\keylist.
```

Since the keys have been defined, they can now be set. In the following, we set only two of the keys:

Example

```
\setkeys[KV]{fam}{align=right,testdim=3cm}
```

The macro `\mp@align` holds the value `\flushright`, while

Example

```
\KV@fam@testdim
```

holds the macros:

Example

```
\def\mp@testdim{#1}
\long\def\f@testdim##1{A test dimension##1\par\bigskip},
```

where ⟨#1⟩ is the value submitted for the key testdim. Try \show\mp@align, \show\KV@fam@testdim, and \show\f@testdim to confirm the above assertions.

The rest of the defined keys can now be set as follows:

Example

```
\setkeys[KV]{fam}{boolvar=true,paperheight,paperwidth,
  textheight,textwidth=6cm}
```

Try \show\ifmp@boolvar to confirm that boolvar is now ⟨true⟩; it was originally set as ⟨false⟩. The macro \KV@fam@paperwidth holds the function \f@paperwidth; \mp@textheight holds the value submitted to key textheight at any instance of \setkeys. By the above \setkeys, only the default values of paperheight, paperwidth, and textheight are presently available.

Instead of using macros to pass key macros and functions, it is also possible to use token registers. An example is provided below:

Example

```
\toks0={\long\def\f@testdim#1{A test dimension #1\par\bigskip}}
\define@keylist{3,boolvar,true,;2,testdim,2cm,\the\toks0}.
```

The advantage of using token registers is that the parameter characters need not be doubled in the token registers, unlike when using macros. The token register \toks0 can be reused as soon as the key testdim is defined. See Section 8.1 for more information on using macros and token registers to pass key functions.

7 Input error

Boolean, toggle and choice keys issue error messages if the key value is not valid, i.e., not in the list of admissible values. The admissible values of boolean and toggle keys are ⟨true⟩ and ⟨false⟩. The valid values of choice keys are set by the user via \VKVS. The default input error is defined by \SKV@inputerr macro to be

Macro

```
\SKV@err{Erroneous value ‘#1’ for key ‘#2’}{%
  Please use the correct value for key ‘#2’.}.
```

\SKV@inputerr can be redefined by the user. It takes two arguments (i.e., value and key).

8 Conditionals in key macros

The TEX conditional primitives \if and \fi cannot appear in the key macro when \define@keylist is being invoked. The reason can be traced to the discussion on page 211 of the TEXBook and the loop used in the skeyval package to define keys. There are three approaches to resolving this problem, and the user can choose anyone he/she prefers.

8.1 Burying conditionals in macros or token registers

Key macros/functions involving conditional operations such as

Example

```
\ifmp@bool \do \fi
```

can be submitted to \define@keylist via macros, as seen above. We give more examples below.

Suppose we want to submit the following:

Example

```
\define@keylist{3,bool,true,\ifmp@bool \do \fi}.
```

The presence of \if and \fi in the argument will trigger an error when TEX is scanning or skipping tokens, and, secondly, because of the loop and conditional used by the skeyval package in defining keys. Neither \protect nor \noexpand is helpful here. One solution is to first define

Example

```
\def\f@bool{\ifmp@bool \do \fi}
```

and then do

Example

```
\define@keylist{3,bool,true,\f@bool},
```

which will execute \f@bool when the key bool is set. Once the key bool has been defined by the above statement, the function \f@bool may be redefined and reused many times, any time, even before the setting of the key bool. It isn't the function \f@bool that is used in defining the key bool, but an internal or meaning of \f@bool, depending on the contents of \f@bool.

As another example, we may do

Example

```
\def\f@abool{\ifmp@abool\def\do####1{%
  \def####1########1{\expandafter\expandafter\expandafter\in@
  \expandafter\expandafter\expandafter{\expandafter####1
  \expandafter}\expandafter{########1}}}\fi}

\define@keylist{3,abool,true,\f@abool}.
```

Token registers (including scratch token registers) can be used here economically instead of macros:

Example

```
\toks0{\ifmp@abool\def\do#1{%
  \def#1##1{\expandafter\expandafter\expandafter\in@
  \expandafter\expandafter\expandafter{\expandafter#1
  \expandafter}\expandafter{##1}}}\fi}

\toks1{\iftog{toggleone}{def\tempa#1{Use #1}}{}}

\define@keylist{3,abool,true,\the\toks0;
  4,toggleone,true,\the\toks1}

\setkeys[KV]{fam}{abool=true,toggleone=true}.
```

You can see the significant reduction in the number of parameter characters when using token registers. The token registers \toks0 and \toks1 can be reused to define many other keys as soon as the keys ⟨abool⟩ and ⟨toggleone⟩ have been defined, even before they are set.

8.2 Using a trick to submit the conditionals

There are two downsides to the above approach of hiding conditionals in macros:

a) The macros have to be defined and, although they can be redefined and reused, they tend to defeat the initial aim of the package, which is to economize on tokens.

b) If the conditionals involve macro definitions as in the above example, the parameter characters have to be doubled in each instance, except when

using token registers.

Suppose we want to define a boolean key mybool with the following key macro:

Example

```
\ifmp@mybool\def\hold##1{\def##1####1{####1}}\fi,
```

where the macro prefix is mp@ and the key family has been defined previously. Then, instead of hiding the conditional in a macro, we can go

Example

```
\define@keylist{3,mybool,true,
  \fif{mp@mybool}\def\hold##1{\def##1####1{####1}}\ffi}.
```

Here we have used \fif{mp@mybool} and \ffi for \ifmp@mybool and \fi, respectively, to hide the latter two from TEX's scanning and skipping mechanism. Please note that \fif{mp@mybool} requires that the argument ⟨mp@mybool⟩ be enclosed in braces. Something like \fifmp@mybool will be interpreted by TEX as undefined control sequence when the key mybool is being set.

We have redefined the \setkeys of the xkeyval package to understand that \fif and \ffi stand for \if and \fi, respectively. The redefined \setkeys command has the same syntax as as in xkeyval package:

Macro

```
\setkeys[⟨prefix⟩]{⟨families⟩}[⟨na⟩]{⟨keys=values⟩}
\setkeys*[⟨prefix⟩]{⟨families⟩}[⟨na⟩]{⟨keys=values⟩}
\setkeys+[⟨prefix⟩]{⟨families⟩}[⟨na⟩]{⟨keys=values⟩}
\setkeys*+[⟨prefix⟩]{⟨families⟩}[⟨na⟩]{⟨keys=values⟩}.
```

The reader who is unfamiliar with the meaning of star and plus signs in the \setkeys command should consult the documentation for the xkeyval package. No errors are produced if any of the sets ⟨prefix⟩, ⟨families⟩, ⟨na⟩, and ⟨keys=values⟩ is empty. In fact, an instruction such as \setkeys[]{}[]{} is completely benign, and so is \setkeys{}{}.

In the case of conditionals starting with \ifcase, a \noexpand before the \ifcase solves the problem:

Example

```
\CKVS{focus}{center,left,right,justified}

\define@keylist{5,focus,center,\noexpand\ifcase\nr\relax
  \def\mp@focus{\centering}\or\def\mp@focus{\flushright}
  \or\def\mp@focus{\flushleft}\or\let\mp@focus\relax\fi
}
```

However, such conditionals may also be buried in macros or token registers.

8.3 Using toggles

Toggle switches, described in Section 4, can also be used to circumvent the problem of matching \if and \fi in difficult circumstances, since toggles aren't TEX primitives. For example, the following works:

Example

```
\define@keylist{4,toggleone,true,
  \iftog{toggleone}{\def\temp{This is defined by a toggle}}{}}.
```

And, as noted in Section 4, toggles are very economical.

9 Checking and redefining keys

9.1 Checking the status of a key

Three mechanism have been introduced in the skeyval package to ascertain the statuses of keys. These are as follows.

Macro

```
\ifkeydefined[⟨prefixes⟩]{⟨families⟩}{⟨key⟩}{⟨true⟩}{⟨false⟩}.
```

This executes ⟨true⟩ if ⟨key⟩ is defined, reserved, or suspended with a prefix in ⟨prefixes⟩ and family in ⟨families⟩; it returns ⟨false⟩ otherwise. This is similar to the xkeyval package's \key@ifundefined, but, apart from reversing the logic of the test, \ifkeydefined loops over prefixes (in addition to looping over families) to locate the key, and also considers reserved and suspended keys as defined. The lists ⟨prefixes⟩ and ⟨families⟩ may contain nil, one or more elements.

Macro

```
\ifkeyreserved[⟨prefixes⟩]{⟨families⟩}{⟨key⟩}{⟨true⟩}{⟨false⟩}
```

This returns ⟨true⟩ if ⟨key⟩ is reserved with a prefix in ⟨prefixes⟩ and family in ⟨families⟩; it returns ⟨false⟩ otherwise. Reserved keys are introduced in Section 10.2.

Macro

```
\ifkeysuspended[⟨prefixes⟩]{⟨families⟩}{⟨key⟩}{⟨true⟩}{⟨false⟩}
```

This executes ⟨true⟩ if ⟨key⟩ is suspended with a prefix in ⟨prefixes⟩ and family in ⟨families⟩; it executes ⟨false⟩ otherwise. Suspended keys are introduced in Section 10.3.

9.2 Unintentional redefinition of keys

The xkeyval package, by default, permits the automatic redefining of keys of the same ⟨prefix⟩ and ⟨family⟩: at the point of defining a new key, the package doesn't, by default, check whether or not the key had been previously defined with the same ⟨prefix⟩ and ⟨family⟩. In some circumstances this can be undesirable, and even dangerous, especially if the same key (of the same ⟨prefix⟩ and ⟨family⟩) is mistakenly redefined with different macros/functions in the same package or across packages. One way to solve this problem is to use xkeyval package's \key@ifundefined command (or the skeyval package's \ifkeydefined) to confirm the status of a key prior to its definition. However, using this command before defining every key can be laborious.

Consider the following two scenarios:

Example

```
\define@key[KV]{fam}{keya}[$\star$]{\def\tempa##1{##1}}
\define@boolkey[KV]{fam}{keya}[true]{%
  \ifKV@fam@keya\def\tempb{#1}\fi}

\setkeys[KV]{fam}{keya=$\textbullet$}
```

Obviously the two definitions of ⟨keya⟩ are valid and will be implemented but the \setkeys command will issue an unintelligible error message, like LaTeX's "You are in trouble here ...". The key ⟨keya⟩ has been defined twice and \setkeys has sought to use its latest definition to set its value, which is incorrect. As mentioned in Section 4.2, the \setkeys command (and friends) of the xkeyval package doesn't know if a key has been redefined in the same ⟨family⟩ and with the same ⟨prefix⟩. At the high level, it doesn't consider the key type: it uses the latest definition of the key to set its value using the key's macro. This is particularly worrisome in the case of toggle keys, since although toggle keys have their own separate namespace, they can easily be confusing (at least to \setkeys) if they have names identical to other keys within the same family and with the same prefix.

9.2.1 Avoiding multiple definitions of same key

For the above reasons, the skeyval package introduces the following commands, which have the same syntaxes as their counterparts from the xkeyval and skeyval packages but which bar the user from repeated definition of keys with identical names within the same ⟨family⟩ and with the same ⟨prefix⟩:

Macro

```
%  The following defines ``ordinary'' keys [the counterpart
%  of \define@key from the xkeyval package]:
\newordkey[⟨prefix⟩]{⟨family⟩}{⟨key⟩}[⟨default⟩]{⟨funtion⟩}
```

```
% Counterpart of \define@cmdkey:
\newcmdkey[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨key⟩}[⟨default⟩]%
  {⟨funtion⟩}

% Counterpart of \define@cmdkeys:
\newcmdkeys[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨keys⟩}[⟨default⟩]

% Counterparts of \define@boolkey:
\newboolkey[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨key⟩}[⟨default⟩]%
  {⟨funtion⟩}
\newboolkey+[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨key⟩}[⟨default⟩]%
  {⟨funtion1⟩}{⟨funtion2⟩}

% Counterpart of \define@compboolkeys:
\newcompboolkeys[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]
  {⟨primary boolean⟩}[⟨default value for primary boolean⟩]
  {⟨secondary boolean⟩}{⟨func for primary boolean⟩}
  {⟨func for secondary boolean⟩}

% Counterparts of \define@togkey:
\newtogkey[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨key⟩}[⟨default⟩]%
  {⟨funtion⟩}
\newtogkey+[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨key⟩}[⟨default⟩]%
  {⟨funtion1⟩}{⟨funtion2⟩}

% Counterpart of \define@comptogkeys:
\newcomptogkeys[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]
  {⟨primary toggle⟩}[⟨default value for primary toggle⟩]
  {⟨secondary toggle⟩}{⟨func for primary toggle⟩}
  {⟨func for secondary toggle⟩}

% Counterparts of \define@choicekey:
\newchoicekey[⟨prefix⟩]{⟨family⟩}{⟨key⟩}[⟨bin⟩]{⟨alt⟩}%
  [⟨default⟩]{⟨funtion⟩}
\newchoicekey*[⟨prefix⟩]{⟨family⟩}{⟨key⟩}[⟨bin⟩]{⟨alt⟩}%
  [⟨default⟩]{⟨funtion⟩}
\newchoicekey+[⟨prefix⟩]{⟨family⟩}{⟨key⟩}[⟨bin⟩]{⟨alt⟩}%
  [⟨default⟩]{⟨funtion1⟩}{⟨funtion2⟩}
\newchoicekey*+[⟨prefix⟩]{⟨family⟩}{⟨key⟩}[⟨bin⟩]{⟨alt⟩}%
  [⟨default⟩]{⟨funtion1⟩}{⟨funtion2⟩}

% Counterpart of \define@keylist:
\newkeylist{⟨key type/id⟩, ⟨key⟩, ⟨key default value⟩,
  ⟨key macro/function⟩; ⟨another set of key specifiers⟩; etc}.
```

We could simply have redefined/modified the legacy key definition commands in the xkeyval package to make it impossible to define keys of the same name in the same family and with the same prefix, but this approach would be unsafe since there are many packages using the xkeyval package and those packages may well have redefined identical keys. Moreover, the legacy key definition commands from the xkeyval package may be needed to redefine a disabled key (see Section 10).

All the commands of the type \newxxxkey are robust and may be used in expansion contexts without fear of premature expansion, although expandable tokens in the definition must be protected.[2]

With the above macros, the following will flag an understandable error message, namely that the key ⟨keya⟩ is about being redefined in the same family ⟨fam⟩ and with the same prefix ⟨KV⟩:

Example

```
\newordkey[KV]{fam}{keya}[$\star$]{\def\tempa##1{##1}}
\newboolkey[KV]{fam}{keya}[true]{%
  \ifKV@fam@keya\def\tempb{#1}\fi}
```

10 Disabling, reserving, suspending, restoring, and removing keys

Besides macros for defining keys, the skeyval package also introduces mechanisms for disabling, reserving, suspending, restoring, and completely removing existing keys.

10.1 Disabling keys

The skeyval package has modified the definition of \disable@keys from the xkeyval package to allow for looping over key prefixes and key families and for bespoke warnings and error messages, without engendering any potential conflict with the legacy \disable@keys. The new command is still called \disable@keys and has the same syntax as the native \disable@keys of the xkeyval package, except that the new command accepts key prefixes (instead of just one prefix) and key families (instead of just one family):

Macro

```
\disable@keys[⟨prefixes⟩]{⟨families⟩}{⟨keys⟩}.
```

[2] We shall refer to keys of the type \newxxxkey as those of category \newkey, and keys of the type \define@xxxkey as those of category \definekey.

Here ⟨prefixes⟩, ⟨families⟩, ⟨keys⟩ are lists of comma-separated entries referring to the keys to be disabled. Each of the lists ⟨prefixes⟩, ⟨families⟩, ⟨keys⟩ may contain nil, one or more elements. If any of the members in ⟨keys⟩ can't be located in ⟨families⟩ and within ⟨prefixes⟩, an informational (not error) message is logged in respect of this member.

The legacy version of \disable@keys (i.e., that of the xkeyval package) is still available via the starred version:

Macro

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

Note that this doesn't accept key prefixes and families, but only one key prefix and only one key family: the \disable@keys command from the xkeyval package can only be used to disable keys with the same ⟨prefix⟩ and from the same ⟨family⟩, but not across prefixes and families.

Any attempt to subsequently set or use a disabled key will prompt the following error message. (The xkeyval package issues a warning in this case.) The error message can be modified by the user, but the names of the controls \SKV@disabledkey@err and \SKV@disabledkey should be retained.

Macro

```
284 \def\SKV@disabledkey@err{%
285   \@latex@error{%
286     Key ⟨key⟩ with prefix ⟨prefix⟩ in family ⟨family⟩
287     was disabled on input line ⟨lineno⟩
288   }{%
289     You can't set or reset ⟨key⟩ at this
290     late stage. Perhaps you're required to set it
291     earlier, in the document's preamble.
292   }%
293 }
```

If the user attempts to disable an undefined key, the xkeyval package issues a fatal error; the skeyval package, on the other hand, issues a warning in the transcript .log file, since the situation isn't fatal to the outcome.

Disabled keys can be redefined with commands in the \definekey category but not with commands in the \newkey category, since a disabled key remains defined: only its macro has been replaced by an error message signifying the disabling of the key.

Note: Reserved and suspended keys can't be disabled, until they are unreserved or restored (see Sections 10.2 and 10.3).

10.2 Reserving and unreserving keys

The xkeyval package bars its users from defining new keys with XKV as a prefix. The skeyval package generalizes this concept via the following three developer macros:

Macro

294 \ReserveKeyPrefixNames{⟨list⟩}

This allows the package developer to bar the future use of names appearing in ⟨list⟩ as key prefixes. The ⟨list⟩, whose members are comma-separated, can be populated by the package developer as required.

Macro

295 \ReserveMacroPrefixNames{⟨list⟩}

This has a similar functionality to \ReserveKeyPrefixNames, but applies to macro prefixes instead of key prefixes.

Macro

296 \ReserveFamilyNames{⟨list⟩}

This applies to family names.

Note: One potential difficulty with the use of these macros is the fact that keys already defined by packages the developer would want to use can't appear in these lists since the lists are scanned both when defining and setting keys.

In addition to the above three commands, the skeyval package also introduces the following command:

Macro

297 \reserve@keys[⟨prefixes⟩]{⟨families⟩}{⟨keys⟩},

where the lists ⟨prefixes⟩, ⟨families⟩, ⟨keys⟩ can contain nil, one or more elements. Defined, reserved and suspended keys can't be reserved.

Reserved keys have to be unreserved with the following command before they can be defined and used:

Macro

298 \unreserve@keys[⟨prefixes⟩]{⟨families⟩}{⟨keys⟩},

where, again, the lists ⟨prefixes⟩, ⟨families⟩, ⟨keys⟩ can contain nil, one or more elements. If a key was not previously reserved, this command will simply issue an informational message in the log file and ignore that key. Incidentally, defined keys and suspended keys can also be unreserved, which is equivalent to removing the keys (see Section 10.4).

10.3 Suspending and restoring keys

For some keys, it might be preferable to temporarily suspend them from a family (rather than disable or remove them) and restore them later. In this way, a key's state and macro can be frozen while the key remains defined.

The syntax for suspending keys is

Macro

299 `\suspend@keys[⟨prefixes⟩]{⟨families⟩}{⟨keys⟩},`

where the lists ⟨prefixes⟩, ⟨families⟩, ⟨keys⟩ can contain nil, one or more elements. A key of particular prefix not previously defined in a family can't be suspended from that family. Similarly, a key previously suspended from a family can't be suspended again (for the second time) from the same family without being first restored in that family.

Suspended keys can be restored to their frozen states (*ex ante* suspension) by the following command:

Macro

300 `\restore@keys[⟨prefixes⟩]{⟨families⟩}{⟨keys⟩}.`

Only keys (with a given prefix) previously suspended from a family can be restored in that family: "unsuspended" keys can't be restored.

10.4 Removing keys

The skeyval package provides for removing keys completely, such that any attempt to set or use a removed key will prompt the error message that the key is undefined in the given family and with the given prefix. The command \key@ifundefined from the xkeyval package and the macro \ifkeydefined from the skeyval package will both identify a removed key as undefined. The syntax for removing keys is:

Macro

301 `\remove@keys[⟨prefixes⟩]{⟨families⟩}{⟨keys⟩}.`

Removed keys can't be restored but can be redefined with the commands in both the \newkey and \definekey categories.

11 Setting keys: list normalization

We have redefined the \setkeys command of the xkeyval package in two respects: firstly to accommodate the use of the \fif and \ffi macros of Section 8.2, and secondly to automatically convert double (or even multiple) com-

mas and equality signs inadvertently submitted by the user into single comma and single equality sign. The following exaggerated example depicts the difficulties that might arise:

Example

```
\define@key[KV]{fam}{width}[1cm]{}
\define@key[KV]{fam}{color}[black]{}
\setkeys[KV]{fam}{width= =2cm, ,,color, == = =,green}
```

Here, the legacy \setkeys will give the value nil to the key width, and the default value of the key color, if it was specified at key definition time, will be given to the key color. Some of the mistakes (especially spurious values without keys) can disrupt a compilation run, while some (multiple commas and equality signs) will not be fatal to compilation but may lead to bizarre results of subsequent calculations. Mistakes of this kind can, surprisingly, be difficult to trace. The extra spaces and multiple commas aren't as serious as the multiple equality signs and values without keys, but we have taken care of all peculiar situations in the new \setkeys. Multiple commas, equality signs, and spaces are now detected and reduced appropriately: that is what we mean by key-value *list normalization*. We have adopted the premise that "`,=`" (comma followed by equal) and "`=,`" (equal followed by comma) are both most likely to mean "`=`" (equal). In the unlikely event that this premise fails, then the user may get tricky errors if he makes this type of mistake: there is perhaps no silver bullet in this regard!

If, for any reason, the user needs to pass keys with "`,=`" and/or "`=,`", then he may separate the comma from the equality sign with `{}`, e.g., as in

Example

```
\setkeys[KV]{fam}{width=2cm,head={},tail=not measured},
```

which shows that the value of the key head is \empty, a valid and better assignment.

12 Miscellaneous macros

This package is predominantly about LaTeX keys and their efficient management, but it also contains many commands for general use, such that a package author may not need to redefine most of them or load some other packages to access those commands. Some of the available commands are described in this section.

Macro

```
\in@tog{⟨subtoken⟩}{⟨token⟩}
```

This is similar to the LaTeX kernel's \in@{⟨subtoken⟩}{⟨token⟩} which tests if ⟨subtoken⟩ is in ⟨token⟩, but this time the returned boolean is the toggle

switch ⟨in@⟩ instead of the kernel's ⟨in@⟩ switch which is used as \ifin@. The toggle ⟨in@⟩ can be used in the following way and in other manners that toggles can be employed:

Example

```
\iftog{in@}{⟨true⟩}{⟨false⟩}.
```

The command \in@tog is robust.

Macro

```
\in@tok{⟨subtoken⟩}{⟨token⟩}
```

Sometimes you want to use the LaTeX kernel's \in@{⟨subtoken⟩}{⟨token⟩} to test if ⟨subtoken⟩ is in ⟨token⟩ irrespective of their catcodes. The robust command \in@tok{⟨subtoken⟩}{⟨token⟩} makes this possible, and eliminates the tokens that would have been necessary if the user was required to first detokenize the two arguments. It returns the same switch \ifin@ as the kernel's \in@{⟨subtoken⟩}{⟨token⟩}.

Macro

```
\SKV@ifdefinable⟨cs⟩{⟨function⟩}
\SKV@for@ifdefinable⟨listcmd⟩{⟨function⟩}
```

LaTeXkernel's \@ifdefinable fills up the hash table and also considers commands that are \relax'ed as defined. Moreover, if the command being tested (⟨cs⟩ in the above example) is definable, the \@ifdefinable macro begins executing ⟨function⟩ while still in the \if ...\fi conditional. The command \SKV@ifdefinable seeks to avoid these problems.

The macro \SKV@for@ifdefinable accepts a comma-separated list ⟨listcmd⟩ of control sequence names whose definability are to be tested. Both commands \SKV@ifdefinable and \SKV@for@ifdefinable are robust.

Macro

```
\SKV@expandargs⟨n⟩⟨function⟩⟨arg1⟩⟨arg2⟩\SKV@nil
```

LaTeX's \@expandtwoargs is often used as a utility macro to expand two arguments ⟨arg1⟩ and ⟨arg2⟩ in order to execute ⟨function⟩. The command \SKV@expandargs accepts up to four expansion types, signified by ⟨n⟩, which runs from 0 to 3:

a) If ⟨n⟩ is 0, then ⟨arg2⟩ is empty and only ⟨arg1⟩ will be expanded before ⟨function⟩ is executed.
b) If ⟨n⟩ is 1, then both ⟨arg1⟩ and ⟨arg2⟩ are nonempty but only ⟨arg2⟩ will be expanded before ⟨function⟩ is executed.

c) When ⟨n⟩ is 2, then both ⟨arg1⟩ and ⟨arg2⟩ are nonempty and both will be expanded before ⟨function⟩ is executed. This is equivalent to LaTeX's \@expandtwoargs.
d) If ⟨n⟩ is 3, then both ⟨arg1⟩ and ⟨arg2⟩ are nonempty but only ⟨arg1⟩ is expanded before ⟨function⟩ is executed.
e) If ⟨n⟩ isn't in the list {0,1,2,3}, then an error message is flagged.

Because ⟨arg2⟩ is delimited, it can be empty. The command \SKV@expandargs can be used to save \expandafter's, but it isn't an all-purpose macro: for example, the \edef it uses may expand too deeply in some cases. Also, care should be exercised when the expanded argument (⟨arg1⟩ and/or ⟨arg2⟩) involve the TeX primitive \if.

Some trivial examples follow:

Example

```
\SKV@expandargs{0}{\def\tempc#1#2}{\def\noexpand##1{##2}}%
  \SKV@nil
\tempc\tempa{aaa}
\tempc\tempb{abcaaabbccbca}
\SKV@expandargs{2}\in@\tempa\tempb\SKV@nil
\show\ifin@
```

Compare these expressions with the chains of \expandafter's in Section 8.1.

Macro

```
\SKV@checkchoice{⟨keyvalue⟩}{⟨altlist⟩}{⟨true⟩}{⟨false⟩}
```

This is the expandable form of xkeyval package's \XKV@checkchoice. It checks if the user-submitted value ⟨keyvalue⟩ of a ⟨key⟩ is in the list ⟨altlist⟩. It executes ⟨true⟩ if ⟨keyvalue⟩ is found in ⟨altlist⟩ and ⟨false⟩ otherwise. Additionally, it returns \val for the expanded value of ⟨keyvalue⟩ and \nr for the numerical order of \val in the list ⟨altlist⟩. If ⟨keyvalue⟩ isn't found in ⟨altlist⟩, then \nr will return −1. If ⟨keyvalue⟩ and ⟨altlist⟩ are buried in macros, the macros are fully expanded before the search for ⟨keyvalue⟩ in the list ⟨altlist⟩ is effected. In that case, \val will hold the expanded form of ⟨keyvalue⟩ and can be used in subsequent computations. Choice keys do accept macros as values, but such values aren't directly suitable for matching against the contents of ⟨altlist⟩. For example, ⟨altlist⟩ may be the set {left,right,center}, but given as a macro \altlist@, while ⟨keyvalue⟩ is given as \def\keyvalue@{center}. Obviously, \keyvalue@ contains one of the elements of ⟨altlist⟩, but choice keys won't know this without the expansion of both \altlist@ and \keyvalue@. This is *raison d'ê tre* of the \SKV@checkchoice macro.

Moreover, \SKV@checkchoice can be used in the definition of non-choice keys. An example follows:

Example

```
\def\altlist@{left,right,center}
\newordkey[KV]{fam}{keya}[true]{%
  \SKV@checkchoice{#1}{\altlist@}{%
    \ifcase\nr\relax
      \edef\tempa##1##2{##1===\val===##2}%
    \or
      \edef\tempa##1##2{##1***\val***##2}%
    \or
      \edef\tempa##1##2{##1+++\val+++##2}%
    \fi
  }{%
    \@latex@error{Wrong value for ‘keya’}\@eha
  }%
}
\def\keyvalue@{center}
\setkeys[KV]{fam}{keya=\keyvalue@}
```

The reader may wish to do \show\tempa to see what \tempa gets upon setting the key keya.

Macro

```
\requirecmd{⟨cs⟩}[⟨number of args⟩][⟨default⟩]%
  {⟨replacement text⟩}
```

This is explained in Section 4.1. If ⟨cs⟩ is already defined, \requirecmd checks if the new and old definitions are identical. If they aren't, a warning message is logged in the transcript file and the new definition is aborted.

Macro

```
\SKV@for@a{⟨list⟩}⟨cmd⟩{⟨function⟩}
\SKV@for@b⟨listcmd⟩⟨cmd⟩{⟨function⟩}
```

A fast for-loop adapted from the xkeyval package to accept general list parsers. Elements of ⟨list⟩ are stored in ⟨cmd⟩, and ⟨function⟩ is executed for each element of ⟨list⟩. The ⟨list⟩, which is populated by comma-separated elements, is not expanded. This accepts a general list parser, dynamically declarable via

Macro

```
\SKV@CommandGenParser{⟨parser⟩}  or
\skvoptions{genparser=⟨parser⟩},
```

instead of just one type of parser ("comma" in the xkeyval package). Also, this uses the more powerful \SKV@ifblank to check whether or not ⟨list⟩ is empty or blank. The command \SKV@for@a is robust, but in expansion contexts, both

⟨cmd⟩ and ⟨function⟩ will need to be somehow protected. In the \SKV@for@b command, ⟨listcmd⟩ is expanded once before the iteration commences. One snag with \SKV@CommandGenParser is that the user must always remember to call it and set the right parser before beginning an iteration, otherwise there might be unpleasant surprises, since a previous call to \SKV@CommandGenParser might have set a parser that is no longer valid.

Macro

```
\SKV@tfor@a⟨cmd⟩{⟨list⟩}{⟨function⟩}
\SKV@tfor@b⟨cmd⟩⟨listcmd⟩{⟨function⟩}
```

The first of these (i.e., \SKV@tfor@a) is equivalent to LaTeX kernel's \@tfor, which loops over ⟨list⟩ token-wise (character or control sequence token), but here we have removed the need for the usual delimitation tokens. Note that ⟨list⟩ is not a comma-separated list! In \SKV@tfor@b, ⟨listcmd⟩ is expanded once before the commencement of the loop. The two commands \SKV@tfor@a and \SKV@tfor@b are both robust.

Macro

```
\SKV@ifstrequal{⟨token1⟩}{⟨token2⟩}{⟨true⟩}{⟨false⟩}
\SKV@oifstrequal{⟨token1⟩}{⟨token2⟩}{⟨true⟩}{⟨false⟩}
\SKV@xifstrequal{⟨token1⟩}{⟨token2⟩}{⟨true⟩}{⟨false⟩}
```

In order to properly test the equality of strings, it may be necessary to remove leading and trailing spaces before the test. Such spaces may have cropped into the strings from input or from pre-processing and may invalidate the test. The macro \SKV@ifstrequal takes care of such situations. It executes ⟨true⟩ if ⟨token1⟩ is equal (character code wise) to ⟨token2⟩, and ⟨false⟩ otherwise. Both ⟨token1⟩ and ⟨token2⟩ are detokenized before the test. The macro \SKV@oifstrequal is similar to \SKV@ifstrequal but first expands its arguments (the two token lists ⟨token1⟩ and ⟨token2⟩) once before the test. The macro \SKV@xifstrequal first expands its arguments fully before the test.

Macro

```
\SKV@ifblank{⟨token⟩}{⟨true⟩}{⟨false⟩}
\SKV@oifblank{⟨token⟩}{⟨true⟩}{⟨false⟩}
\SKV@xifblank{⟨token⟩}{⟨true⟩}{⟨false⟩}
```

These macros test if the argument is blank or not. The first of these is from ifmtarg package. \SKV@oifblank expands its argument once before the test, while \SKV@xifblank expands its argument fully before the test.

Macro

```
\SKV@ifstrempty{⟨token⟩}{⟨true⟩}{⟨false⟩}
\SKV@oifstrempty{⟨token⟩}{⟨true⟩}{⟨false⟩}
\SKV@xifstrempty{⟨token⟩}{⟨true⟩}{⟨false⟩}
```

These yield ⟨true⟩ if ⟨token⟩ is empty, and ⟨false⟩ otherwise. In the macro \SKV@ifstrempty, ⟨token⟩ isn't expanded before the test; in the command \SKV@oifstrempty, ⟨token⟩ is expanded once before the test; in the command \SKV@xifstrempty, ⟨token⟩ is fully expanded before the test.

Macro

352 \SKV@expandox{⟨cs⟩}

This expands its argument ⟨cs⟩ once and forbids further expansion.

Macro

353 \SKV@expandnameox{⟨name⟩}

This is similar to \SKV@expandox but accepts control sequence name ⟨name⟩ instead of control sequence.

Macro

354 \SKV@expandtx{⟨cs⟩}

This expands its argument ⟨cs⟩ twice and forbids further expansion.

Macro

355 \SKV@expandnametx{⟨name⟩}

This is similar to \SKV@expandtx but accepts control sequence name instead of control sequence.

Macro

356 \@afterpackageloaded{⟨package⟩}{⟨code⟩}

This executes ⟨code⟩ only after the package ⟨package⟩ has been loaded. This has been optimized from the afterpackage package to avoid filling up the hash table with hooks that are relax'ed or indeed undefined.

Macro

357 \SKV@newlet{⟨cs1⟩}{⟨cs2⟩}

This assigns ⟨cs2⟩ to ⟨cs1⟩ if ⟨cs2⟩ exists and if ⟨cs1⟩ isn't already defined, otherwise an error is flagged.

Macro

358 \SKV@newtoks{⟨toks⟩}

This provides a new token register ⟨toks⟩ if the register didn't already exist, otherwise an error is flagged.

Macro

```
\SKV@newcount{⟨counter⟩}
```

This provides a new counter register ⟨counter⟩ if the register didn't already exist, otherwise an error is flagged.

Macro

```
\SKV@newdimen{⟨dimen⟩}
```

This provides a new dimension register ⟨dimen⟩ if the register didn't already exist, otherwise an error is flagged.

Macro

```
\SKV@numdef{⟨num⟩}{⟨expression⟩}
\SKV@numnamedef{⟨num⟩}{⟨expression⟩}
\SKV@numgdef{⟨num⟩}{⟨expression⟩}
\SKV@numnamegdef{⟨num⟩}{⟨expression⟩}
```

\SKV@numdef defines ⟨num⟩ from ⟨expression⟩ using ε-TEX's \numexpr. It is similar to etoolbox's \numdef but is defined by a counter expression instead of \edef (which etoolbox uses). If ⟨num⟩ is previously undefined, it is first initialized before the expression is built. The difference between \numdef and \SKV@numdef is that if you do \numdef\x{1+2+3}, you can use \x without, depending on context, prefixing it with \the or \number. \SKV@numdef, on the other hand, will require \the or \number. Also, \SKV@numdef includes two tests of its arguments.

\SKV@numnamedef is the same as \SKV@numdef but takes a control sequence name instead of a control sequence.

The macros \SKV@numgdef and \SKV@numnamegdef, unlike \SKV@numdef and \SKV@numnamedef, effect global assignments and thus can escape local groups.

Macro

```
\SKV@dimdef{⟨dimen⟩}{⟨expression⟩}
\SKV@dimnamedef{⟨dimen⟩}{⟨expression⟩}
\SKV@dimgdef{⟨dimen⟩}{⟨expression⟩}
\SKV@dimnamegdef{⟨dimen⟩}{⟨expression⟩}
```

\SKV@dimdef defines ⟨dimen⟩ from ⟨expression⟩ using ε-TEX's \dimexpr. It is similar to etoolbox's \dimdef but is defined by a dimension expression instead of \edef. If ⟨dimen⟩ is previously undefined, it is first initialized before the expression is built. The difference between \dimdef and \SKV@dimdef is that if you do \dimdef\x{1pt+2mm+3cm}, you can use \x without prefixing it with \the. \SKV@dimdef, on the other hand, will require \the.

The macros \SKV@dimgdef and \SKV@dimnamegdef, unlike \SKV@dimdef and \SKV@dimnamedef, effect global assignments.

Macro

```
\SKV@newnamedef{⟨name⟩}{⟨definition⟩}
\SKV@newnamegdef{⟨name⟩}{⟨definition⟩}
\SKV@newnameedef{⟨name⟩}{⟨definition⟩}
\SKV@newnamexdef{⟨name⟩}{⟨definition⟩}
```

These turn ⟨name⟩ into a control sequence if it wasn't already defined. If it is already defined, an error message is flagged. These derive from a concept based on that of \newcommand, but (i) \relax'ed commands are considered undefined in this regard, and (ii) these commands retain the powerful machinery of plain TEX.

Macro

```
\SKV@namedef{⟨name⟩}{⟨definition⟩}
\SKV@namegdef{⟨name⟩}{⟨definition⟩}
\SKV@nameedef{⟨name⟩}{⟨definition⟩}
\SKV@namexdef{⟨name⟩}{⟨definition⟩}
```

These turn ⟨name⟩ into a control sequence whether or not the control was already defined. No error or warning messages are issued.

Macro

```
\nameletcs{⟨name⟩}{⟨cs⟩}
\csletname{⟨cs⟩}{⟨name⟩}
\nameletname{⟨name⟩}{⟨name⟩}
```

These perform \let assignments if the second argument is defined, otherwise an error message is flagged. ⟨cs⟩ means a control sequence.

Macro

```
\@nameletcs{⟨name⟩}{⟨cs⟩}
\@csletname{⟨cs⟩}{⟨name⟩}
\@nameletname{⟨name⟩}{⟨name⟩}
```

These perform \let assignments whether or not the second argument is defined. If the second argument is undefined, the hash table is not filled.

Macro

```
\SKV@nameuse{⟨name⟩}
```

This is similar to LATEX's \@nameuse but returns \@empty if ⟨name⟩ is undefined. The idea derives from the etoolbox package.

Macro

```
\SKV@ifdef{⟨cs⟩}{⟨true⟩}{⟨false⟩}
\SKV@ifnamedef{⟨name⟩}{⟨true⟩}{⟨false⟩}
\SKV@ifundef{⟨cs⟩}{⟨true⟩}{⟨false⟩}
\SKV@ifnameundef{⟨name⟩}{⟨true⟩}{⟨false⟩}
```

These use ε-TEX's facilities to test the existence of the control sequence ⟨cs⟩ or ⟨name⟩. \relax'ed commands are considered undefined in this regard. These are based on similar concepts from the etoolbox package.

Macro

```
\SKV@findescape{⟨arg⟩}{⟨true⟩}{⟨false⟩}
```

This returns ⟨true⟩ if ⟨arg⟩ starts with the \escapechar, and ⟨false⟩ otherwise. The \escapechar is locally made equal to 92 before the test.

Macro

```
\SKV@undef{⟨cs⟩}
\SKV@gundef{⟨cs⟩}
\SKV@nameundef{⟨name⟩}
\SKV@namegundef{⟨name⟩}
```

These undefine the macro ⟨cs⟩ or control sequence name ⟨name⟩ such that TEX will subsequently consider them undefined. The macros ⟨\SKV@gundef⟩ and ⟨\SKV@namegundef⟩ undefine their argument globally.

Macro

```
\SKV@aftergroup{⟨code⟩}
\SKV@aftergroup*{⟨code⟩}
\SKV@afterassignment{⟨code⟩}
\SKV@afterassignment*{⟨code⟩}
```

These execute the arbitrary ⟨code⟩ after a group or assignment. The starred versions fully expand ⟨code⟩ before the assignment (or before exiting the group). These are similar to \AfterGroup and \AfterAssignment of etextools package but they don't accumulate the group and assignment counters indefinitely: the counters are initialized after each group or each assignment.

Some examples follow:

Example

```
\def\aa{aaa}\def\bb{bbb}\def\xx{xxx}\def\yy{yyy}
\begingroup
  \SKV@aftergroup{\par\aa***\bb}%
  \SKV@aftergroup{\par\bb***\aa}%
  \begingroup
```

```
    \SKV@aftergroup{\par\xx+++\yy}%
    \SKV@aftergroup{\par\yy+++\xx}%
  \endgroup
\endgroup

\let\gobblex\@firstofone
\def\protected@funnydef{%
   \let\@@protect\protect
   \let\protect\@unexpandable@protect
   \SKV@afterassignment{\restore@protect\let\gobblex\@gobble}%
   \edef
}
```

Macro

```
\SKV@apptomacro@a⟨cs⟩⟨content⟩
\SKV@apptomacro@b⟨cs⟩⟨content⟩
\SKV@gapptomacro@a⟨cs⟩⟨content⟩
\SKV@gapptomacro@b⟨cs⟩⟨content⟩
```

This appends ⟨content⟩ to ⟨cs⟩. If ⟨cs⟩ was previously undefined, it is initialized with ⟨content⟩. \SKV@apptomacro@a doesn't expand ⟨content⟩ before appending it to ⟨cs⟩, while \SKV@apptomacro@b expands ⟨content⟩ once before appending it's contents to ⟨cs⟩. The macros \SKV@gapptomacro@a and \SKV@gapptomacro@b have global effect, which can escape local groups. Except for the initialization of undefined ⟨cs⟩, \SKV@gapptomacro@a is equivalent to LaTeX's \g@addto@macro.

Macro

```
\SKV@preptomacro@a⟨cs⟩⟨content⟩
\SKV@preptomacro@b⟨cs⟩⟨content⟩
\SKV@gpreptomacro@a⟨cs⟩⟨content⟩
\SKV@gpreptomacro@b⟨cs⟩⟨content⟩
```

These perpend ⟨content⟩ to ⟨cs⟩. If ⟨cs⟩ was previously undefined, it is initialized with ⟨content⟩. \SKV@preptomacro@a doesn't expand ⟨content⟩ before prepending it to ⟨cs⟩, while \SKV@preptomacro@b expands ⟨content⟩ once before prepending it's contents to ⟨cs⟩. The macros \SKV@gapptomacro@a and \SKV@gapptomacro@b have global effect.

Macro

```
\InputFileOnce[⟨path⟩]{⟨filename⟩}
```

This inputs ⟨filename⟩ on ⟨path⟩ but only once. The argument ⟨path⟩ is optional and its default value is the current/document's directory. If ⟨filename⟩ had previously been read, an error message is flagged and the input is aborted.

If ⟨filename⟩ doesn't exist on ⟨path⟩, an opportunity is given to the user to type in the correct file name on the screen, or continue with the LaTeX pass without inputting the file. If the user wants to input ⟨filename⟩ more than once in one document, then he should set \SKV@inputonce to ⟨false⟩ before the call to \InputFileOnce, either by simply issuing \SKV@inputoncefalse or via \skvoptions:

Example

```
423 \skvoptions{inputonce=false}.
```

\SKV@inputonce can, of course, be toggled between ⟨false⟩ and ⟨true⟩.

Macro

```
424 \AtStartOfDocument{⟨code⟩}
```

This is the expansion-context version of the hook \AtBeginDocument. Sometimes you want to do something like

Macro

```
425 \edef\tempa{\AtBeginDocument{%
426   \def\noexpand\tempb{come with me}}},
```

which is impossible without a \noexpand before \AtBeginDocument. With \AtStartOfDocument the protection is not necessary.

Two more expansion-context document hooks are provided below:

Macro

```
427 \BeforeStartOfDocument{⟨code⟩}
428 \AfterStartOfDocument{⟨code⟩}.
```

These mimic the \AtEndPreamble and \AfterEndPreamble hooks from the etoolbox package.

13 Version history

This package was called the keyreader package until version 0.5, when the name became a misnomer.

a) Version 0.1 [01/01/2010]
   i) Provided machinery for reading multiple keys of all kinds from just one command.
   ii) Introduced complementary boolean keys.
b) Version 0.2 [10/01/2010]
   i) Corrected a bug.
c) Version 0.3 [20/01/2010]

i) Introduced toggles and toggle keys.

d) Version 0.4 [01/02/2010]
   i) Introduced complementary toggle keys.

e) Version 0.5 [23/02/2010]
   i) Provided facilities for avoiding repeated definition of same key (when desired).
   ii) Provided mechanisms for disabling, reserving, unreserving, suspending, restoring, and removing keys after they have been defined.
   iii) Redefined a few of `xkeyval` package's macros.
   iv) Included some developer macros.