The skeyval Package[☆,★]

Version 0.71

Ahmed Musa ✉
Preston, Lancashire, UK

13th May 2010

Abstract

This package supplements the xkeyval package. It introduces toggle keys and complementary (bipolar and unipolar) native-boolean and toggle-boolean 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. The package introduces the notion of "user-value keys" and provides facilities for managing those keys. The pointer mechanisms of the xkeyval, which were only available at key setting time, are now invocable at key definition. Some other general-purpose developer macros and hooks are provided by the package.

[☆]The skeyval package was formerly called the keyreader package until version 0.5. The keyreader is now obsolete and no longer supported.

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

Contents

1 Motivation

Toggle booleans were introduced by the etoolbox package and have proved attractive mainly for two reasons: unlike the legacy/native 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

```
% Knuth/native switch:
\newif\ifmyboolean     →  3 separate commands:
                          \ifmyboolean \mybooleantrue
                          \mybooleanfalse

% Toggle switch:
\newtog{myboolean}     →  only 1 command and no clash with
                          commands in other namespaces
```

Note: A toggle is also a boolean or switch. We refer to it here as *toggle* or *toggle switch* or *toggle boolean*. The term *Knuth/native switch* is reserved here for TEX's legacy boolean or switch.

The xkeyval package can't be used to define and set toggle keys. The present package provides facilities for defining and setting toggle keys.

Secondly, the xkeyval package can't be used to define and set complementary (bipolar and unipolar) keys, which can be handy in the case of native boolean and toggle keys. Complementary bipolar keys are mutually exclusive keys, i.e., they never assume the same state of a two-valued logic, and they switch states automatically, depending on the state of any one of them. So whenever one of them assumes one of the two states of a two-valued logic, the other one automatically switches its state from whatever state it was previously, such that the two are never in the same state. Simple examples of complementary bipolar keys would be the options draft and final in a document; they are mutually exclusive. Complementary bipolar keys carry equal charge, i.e., each one can equally toggle the other with identical propensity, but the one that represents the default state of a two-valued logic is usually considered the primary, while the other is secondary. In many document classes, for instance, final is considered a default document option and draft would have to be explicitly selected by the user to toggle final to false.

Complementary unipolar keys, on the other hand, are mutually inclusive, i.e., both are always in the same state of a two-valued logic: when one is switched to a particular state, the complement too is automatically toggled to the same state. An example of complementary unipolar keys would be the options hyperref and microtype in a package or class file. When hyperref package is loaded, we may want microtype package loaded as well, and vice versa. Complementary unipolar keys, like bipolar keys, also carry equal charge, i.e., one can toggle the other with equal propensity, but to the same state for the two keys. For a pair of unipolar keys, the primary key is the one that is associated with the default state.

The present package introduces these concepts of bipolar and unipolar keys and additionally permits the submission of individual/different custom key macros to complementary (bipolar and unipolar) native boolean and toggle keys. Bi-unipolar keys, which combines the two concepts of bipolar and unipolar keys, are introduced as well.

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/menu 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 has a philosophy akin to the \newcommand concept in LaTeX.

The package also provides facilities for disabling, suspending, restoring and removing keys across multiple families of keys. The pointer system of the xkeyval package is also extended by the skeyval package, and the notion of "user-value" keys is introduced and operationalized.

Normally when setting keys, the xkeyval package terminates with an error message if any key in the list of keys to be set is currently undefined or unknown. This means that if you have a set of keys to set simultaneously and you misspell a few of them, you would have to make several passes to get all the keys right. The skeyval package takes a different approach. For each call to \setkeys, it saves the names of undefined keys (and their associated families and prefixes) in a macro and issues an error message (together with the contents of that macro) if at the end of the call that macro is non-empty. Such undefined keys are not set in the pass. In this way, you can correct all wrongly entered keys in one go, rather than repeat the pass for each of them.

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

2 Brace stripping by the xkeyval package

The xkeyval package strips off up to three levels of braces in the value part of the key-value pair: one by using the keyval package's leading and trailing space removal command and two in internal parsings (at some known commands). The keyval strips off only two levels of braces: one in using its space removal routine and one in internal parsings. The kvsetkeys package strips off only one level of braces. The matter of these differences has not yet been shown to have serious implications for existing or new packages, although it is apparently of concern to the keycommand package.

The skeyval package internally uses a space removal scheme that doesn't strip off braces from its arguments, but since the present package relies, to a good extent, on the engine of the xkeyval package, the brace stripping effect has remained. We have had to redefine a significant number of the internal macros of the xkeyval package, but getting rid of the brace-stripping issue would require even more substantial revision of the internal macros of the xkeyval package. It can easily be solved but there is currently no sufficient reason to do so. If, odd enough, you want at least one level of braces to persist in the value part of the

key-value pair throughout parsing, you simply surround the value with four levels of braces in the \setkeys command. If the users of this, or the xkeyval package, feel that this issue is of significant concern, then it can be addressed in the future.

3 The skeyval package options

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 user options and their default values are

Macro

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

The ⟨keyparser⟩ is the separator between the keys in the key list to be defined in one go (see Subsection 7.2). 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}
```

These options are explained in more detail in subsequent sections.

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.

The skeyval package issues a fatal error if it is loaded before (or run without) \documentclass.

4 Complementary native-boolean keys

4.1 Bipolar native-boolean keys

As mentioned in Section 1, complementary bipolar keys are keys that depend inversely on each other: when one of them is in a particular state of a two-valued logic, the other one automatically assumes the opposite or complementary state.

For each pair of bipolar keys, one is normally assumed to be the primary key and the other the secondary. The primary boolean key will usually represent the default state of a two-valued logic. Whenever one bipolar key (primary or secondary) is true, its complement is automatically set false; and vice versa: when one bipolar key (primary or secondary) is false, its complement is automatically set true. Generally, the transition of the state of a key from negative (false) to positive (true) is associated with the execution of the key's macro.

The syntax for creating bipolar native-boolean keys is

Macro

```
\define@biboolkeys[⟨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 may need to be protected. When the user doesn't supply the ⟨keyprefix⟩ and/or ⟨macroprefix⟩, the package will use ⟨KV⟩ and ⟨mp@⟩, respectively. When the default value for the primary boolean is not supplied, the package will use true. Infinite loops, which are possible in back-linked key settings, are avoided in the skeyval package. The machinery of the xkeyval package, such as \setkeys, \presetkeys, \savekeys, \savevalue, \usevalue, etc., are all applicable to all complementary keys.

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

Example

```
\define@biboolkeys[KV]{fam}[mp@]{draft}[true]{final}%
{%
  \ifmp@draft
    \def\noneofone##1{}%
  \else
    \def\oneofone##1{##1}%
  \fi
}{%
  \ifmp@final
    \def\noneoftwo##1##2{}%
  \else
    \def\oneoftwo##1##2{##1}%
  \fi
}
```

The key prefix (default KV), macro prefix (default mp@), key macros (no default), and the default value of the primary boolean (true) can all be empty:

Example

```
\define@biboolkeys{fam}{draft}{final}{}{}.
```

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

Example

```
33 \setkeys[KV]{fam}{draft=true or on}

34 \setkeys[KV]{fam}{final=true or on}
```

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

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

Note: In the skeyval package, the acceptable values for native-boolean and toggle-boolean keys are true, on, false, and off. Toggle-booleans (Subsection 5.1) and switches (Subsection 7.4.3) also accept these values. The value on is synonymous with true, whilst off is an alias for false. Being merely aliases, both on and off don't increase the number of commands per switch: the number remains three.

The skeyval package has the \NewIfs macro which defines five commands per switch:

Macro

```
36 \NewIfs[⟨optional prefix⟩]{⟨boolean list⟩}[⟨optional state⟩]
```

This provides, for each member of the comma-separated list ⟨boolean list⟩, a new native-boolean register if the register didn't already exist, otherwise an error is flagged. Each member of ⟨boolean list⟩ is prefixed with ⟨prefix⟩ upon definition. The optional ⟨state⟩ can be either true, false, on or off; the default is false. The state on is synonymous with true, whilst off is equivalent to false. After declaring

Example

```
37 \NewIfs[bool]{a,b}[true]
```

you can say \ifboola, \boolaon and \boolaoff, just like you would normally do \boolatrue and \boolafalse. The disadvantage of \NewIfs is that it defines five commands per switch; it should therefore be employed only in special circumstances.

4.2 Unipolar native-boolean keys

Unipolar boolean keys are two keys that are always in the same state: when one is true (or false), the other one is also true (or false). In this regard, the key macro is always executed when a key transits to the "true" state. The syntax for creating unipolar native-boolean keys is exactly as that for defining bipolar native-boolean keys:

Macro

```
\define@uniboolkeys[⟨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 may need to be protected. Again, if the user doesn't supply the ⟨keyprefix⟩ and/or ⟨macroprefix⟩, the package will use ⟨KV⟩ and ⟨mp@⟩, respectively. When the default value for the primary boolean is not supplied, the package will assume it to be true.

The following example constructs two unipolar native-boolean keys:

Example

```
\NewToks[temptoks]{a,b}

\define@uniboolkeys[KV]{fam}[mp@]{pdfmode}[true]{microtype}%
{%
  \ifmp@pdfmode
    \temptoksa{Yes, in 'pdfmode'}%
  \else
    \temptoksa{No, not in 'pdfmode'}%
  \fi
}{%
  \ifmp@microtype
    \temptoksb{Yes, 'microtype' loaded}%
  \else
    \temptoksb{No, 'microtype' not loaded}%
  \fi
}
```

4.3 Biunipolar native-boolean keys

Biunipolar keys are the generalized forms of bipolar and unipolar boolean keys, with one important restriction: unlike bipolar and unipolar keys, biunipolar keys have no symmetrical relationships. That is to say that the relationship between a pair of biunipolar keys is entirely determined by the primary key. A pair of biunipolar boolean keys possess only one of the following four types of relationship:

a) Unipolar property: When the primary key is false, it sets the secondary key to false (⋆ +-form of biunipolar keys). The secondary key macro isn't executed.
b) Bipolar property: When the primary key is false, it sets the secondary key to true (⋆-form). The secondary key macro is executed.
c) Bipolar property: When the primary key is true, it sets the secondary key to false (+-form). The secondary key macro isn't executed.
d) Unipolar property: When the primary key is true, it sets the secondary key to true (unsigned form). The secondary key macro is executed.

Of course, the primary key can be true only after it has been set.

The syntax for establishing biuni boolean keys is exactly like that for creating other complementary boolean keys, except for the optional ⋆ and + signs:

Macro

```
\define@biuniboolkeys*+[⟨keyprefix⟩]{⟨family⟩}[⟨macroprefix⟩]
  {⟨primary boolean⟩}[⟨default value for primary boolean⟩]
  {⟨secondary boolean⟩}{⟨macro for primary boolean⟩}
  {⟨macro for secondary boolean⟩}
```

As an example, consider the arbitrary package or class options review and preprint. The option preprint can automatically toggle review to true, but possibly not vice versa: not every preprint is a manuscript for review. This is depicted below:

Example

```
\define@biuniboolkeys[KV]{fam}[mp@]{preprint}[true]{review}{%
  % No key macro for preprint; otherwise, it would
  % have come in here.
}{%
  \ifmp@review
    \SKV@BeforeDocumentStart{%
      \linespread{1.5}\selectfont
      \def\banner{\fbox{\textit{This is a review document}}}%
    }%
  \else
    \SKV@BeforeDocumentStart{\let\banner\@empty}%
  \fi
}
```

As another example, consider the following biuni keys, each with its own macro:

Example

```
\define@biuniboolkeys*+[KV]{fam}[mp@]{brother}[true]{sister}{%
  \ifmp@brother
    \def\mybrother{Hamilton}%
  \fi
}{%
  \ifmp@sister
    \SKV@AfterDocumentStart{\def\mysister{Kate}}%
  \else
    \SKV@AfterDocumentStart{\let\mysister\@gobble}%
  \fi
}

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

5 Toggle booleans and toggle keys

In the following Subsections 5.1 to 5.2 we define toggle booleans/switches and use them to introduce toggle-boolean keys.

5.1 Toggle booleans

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. The internal control sequences and user interfaces of the two packages are, however, different, thus avoiding clashes.

All the commands in this section are robust and can be used in expansion or moving contexts without fear of premature expansion, but fragile arguments would need to be protected in those settings.

Macro

```
86 \deftog[⟨optional prefix⟩]{⟨toggle⟩}[⟨optional state⟩]
```

This defines a new ⟨toggle⟩, prefixed with ⟨prefix⟩ upon definition, whether or not ⟨toggle⟩ (with its prefix) is already defined. If ⟨toggle⟩ is already defined, a warning message is logged in the transcript file (if the package option verbose is selected) and the new definition is effected. The optional ⟨state⟩ can be either true, false, on or off.

Macro

```
87 \newtog[⟨optional prefix⟩]{⟨toggle⟩}[⟨optional state⟩]
```

This defines a new ⟨toggle⟩, prefixed with ⟨prefix⟩ upon definition, if ⟨toggle⟩ (with its prefix) is not already defined; otherwise the package issues a fatal error. The optional ⟨state⟩ can be either true, false, on or off.

You can define a set of toggles by the following command:

Macro

```
88 \NewTogs[⟨optional prefix⟩]{⟨toggles⟩}[⟨optional state⟩],
```

where ⟨toggles⟩ is a comma-separated list. Each member of ⟨toggles⟩ is prefixed with ⟨prefix⟩ upon definition. The optional ⟨state⟩ can be either true, false, on or off. For example, we may define new toggles togx, togy, togz by the following:

Example

```
89 \NewTogs[tog]{x,y,z}[true]
```

Macro

```
90 \providetog{⟨toggle⟩}
```

This defines a new ⟨toggle⟩ if ⟨toggle⟩ is not already defined. If ⟨toggle⟩ is already defined, the command does nothing. Please note that there are no tog prefix here, and that the default state is false.

Macro

```
91 \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 (if the package option verbose is selected) \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, if the package option verbose is selected, 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. Note that there are no tog prefix here, and that the default state is false.

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 14).

Macro

```
92 \settog{⟨toggle⟩}{⟨true | false | on | off⟩}
```

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

Macro

```
93 \togon{⟨toggle⟩}
94 \togtrue{⟨toggle⟩}
```

These set ⟨toggle⟩ to ⟨true⟩ or on. They will issue an error if ⟨toggle⟩ wasn't previously defined.

Macro

```
95 \togoff{⟨toggle⟩}
96 \togfalse{⟨toggle⟩}
```

These set ⟨toggle⟩ to ⟨false⟩ or off. They will issue an error if ⟨toggle⟩ wasn't previously defined.

Macro

```
97 \iftogon{⟨toggle⟩}{⟨true⟩}{⟨false⟩}
98 \iftogtrue{⟨toggle⟩}{⟨true⟩}{⟨false⟩}
```

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

Macro

```
\iftogoff{⟨toggle⟩}{⟨not true⟩}{⟨not false⟩}
\iftogfalse{⟨toggle⟩}{⟨not true⟩}{⟨not false⟩}
```

These behave like \iftogon and \iftogtrue but reverse the logic of the test. They will issue an error if ⟨toggle⟩ wasn't previously defined.

5.2 Toggle-boolean keys

The user interfaces for defining toggle-boolean keys is exactly like those for native-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-boolean 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 native-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 Subsection 9.2. 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 Subsection 9.2).

The user interfaces for defining toggle keys are

Macro

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

\define@togkey+[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨key⟩}[⟨default⟩]%
  {⟨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:

Example

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

\define@togkey+{fam}{shadow}{%
  \iftogon{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 7 for how to define multiple keys with custom functions.

As an example of defining multiple toggle keys, consider

Example

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

This is an abbreviation for

Example

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

Now we can do

Example

```
\define@togkey{fam}[my@]{book}{%
  \iftogon{my@book}{\setkeys[KV]{fam}{togc=true}}{}%
}
\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

```
\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

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

6 Complementary toggle keys

6.1 Bipolar toggle-boolean keys

Similar to complementary native-boolean keys of Section 4, the skeyval package introduces facilities for creating complementary (bipolar, unipolar and biunipolar) toggle keys. The syntax for defining bipolar toggle keys is identical to that for defining bipolar native-boolean keys:

Macro

```
\define@bitogkeys[⟨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 the default value for the primary toggle-boolean is not supplied, the package will use true. When one of the bipolar toggle keys (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 bipolar toggle keys ⟨xdraft⟩ and ⟨xfinal⟩ with different key macros:

Example

```
\define@bitogkeys[KV]{fam}[mp@]{xdraft}[true]{xfinal}%
{\def\gobble##1{}}{\def\firstofone##1{##1}}
```

The key prefix (default KV), macro prefix (default mp@), key macros (no default), and the default value of the primary boolean (default true) can all be empty:

Example

```
\define@bitogkeys{fam}{xdraft}{xfinal}{}{}.
```

The defined bipolar 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⟩.

Note: Toggle keys may easily be confused with the conventional boolean keys, especially at the time of key setting. It is, therefore, always safer to use the syntaxes in Subsection 9.2 for defining keys; they avoid interference between new and existing keys.

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) native-boolean keys in Section 4—because they share the same family ⟨fam⟩ and prefix ⟨KV⟩. The names draft and final could safely be used as toggles only if the family ⟨fam⟩ or prefix ⟨KV⟩ is changed. See Subsection 9.2.1 for further details.

6.2 Unipolar toggle-boolean keys

The syntax for defining unipolar toggle keys is exactly the same as that for defining bipolar toggle keys:

Macro

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

Here too, if the user doesn't supply the ⟨keyprefix⟩ and/or ⟨macroprefix⟩, the package will use ⟨KV⟩ and ⟨mp@⟩, respectively. When the default value for the primary toggle key is not supplied, the package will use true.

Example

```
\define@unitogkeys[KV]{fam}[mp@]{draft}[true]{final}%
  {\def\x##1{}}{\def\y##1{##1}}
\setkeys[KV]{fam}{draft=true}% ``final'' becomes ``true'' here.
```

6.3 Biunipolar toggle-boolean keys

The interface for creating biuni toggle keys is exactly like that for creating other complementary boolean keys, except for the optional ⋆ and + signs. The interface is as follows (the meaning of the optional ⋆ and + is given in Subsection 4.3):

Macro

```
\define@biunitogkeys*+[⟨keyprefix⟩]{⟨family⟩}[⟨macroprefix⟩]
  {⟨primary boolean⟩}[⟨default value for primary boolean⟩]
  {⟨secondary boolean⟩}{⟨macro for primary boolean⟩}
  {⟨macro for secondary boolean⟩}
```

Example

```
\define@biunitogkeys+[KV]{fam}[mp@]{preprint}[true]{review}{%
  \iftogon{mp@preprint}{%
    \def\banner{\fbox{\textsf{This is a preprint copy}}}%
  }{}%
}{%
  \iftogon{mp@review}{%
    \SKV@AtDocumentStart{\linespread{1.5}\selectfont}%
    \def\banner{\fbox{\textit{This is a review article}}}%
  }{%
    \let\banner\@empty
  }%
}
\setkeys[KV]{fam}{preprint=false or true}
```

7 Defining multiple keys of all genres by one command

The user 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}
```

Here, there are five key types: 1 (ordinary key), 2 (command key), 3 (native-boolean key), 4 (toggle-boolean key), and 5 (choice/menu key). The key types can be indicated either in numeral format (`1` to `5`) or in alphabetic format (`ord`, `cmd`, `bool`, `tog`, `choice`, `menu`). "Choice" and "menu" keys imply the same thing (key type 5): the user can pick the name he prefers. The key and its attributes are separated by commas; they constitute one "object" or "instance". The objects are separated by the ⟨`keyparser`⟩, which is the semicolon in the above example. The ⟨`keyparser`⟩ is a package option and can be changed dynamically.

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 Subsection 9.2. The machinery of Subsection 9.2 can be utilized to safely define new keys without the fear of inadvertently redefining an existing key within the same family and with the same key prefix.

Keys defined by the tools in this section are amenable to management by pointers of Section 12.

7.1 Choice key values

The \choicekeyvalues macro is provided for defining 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 via \define@keylist. Its syntax is

Macro

```
\choicekeyvalues*[⟨prefix⟩][⟨family⟩]{⟨key⟩}{⟨list⟩},
```

where ⟨list⟩ is a comma-separated list of admissible key values. Unless the key prefix or family changes, the unstarred variant of \choicekeyvalues wouldn't allow you to set two \choicekeyvalues for the same choice key. The starred variant \choicekeyvalues*, on the other hand, allows you to overwrite admissible choice values for a key within a specified family and with the given key prefix. The arguments ⟨prefix⟩ and ⟨family⟩ are optional, provided the key prefix and family have been specified before calling \choicekeyvalues, using \skvoptions. If ⟨prefix⟩ and ⟨family⟩ are not given, the prevailing key prefix and key family are used internally by \choicekeyvalues to build distinct alternate values list for the choice key. 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 \choicekeyvalues. It doesn't matter which choice key first gets a \choicekeyvalues.

To further save tokens, the macro \choicekeyvalues may be abbreviated by \CKVS. It has to be provided for each choice key that is being defined.

For example, if we want to define two choice keys align and shootingmodes, then before the call to \define@keylist, we have to set

Example

```
\CKVS{align}{center,right,left,justified}
\CKVS{shootingmodes}{portrait,indoor,foliage,underwater}
```

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}.
```

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

7.2 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}
```

Further, suppose we wish to define a set of keys {color,angle,scale,align}. The key color will be defined as ordinary key, the keys 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 key color isn't associated with any macro. Then we can do:

Example

```
% We use space freely in these examples for the sake
% of illustration:
\def\someweather{windy}
\CKVS{align}{center,right,left,justified}
\CKVS{weather}{sunny,cloudy,lightrain,heavyrain,snow,
  sleet,\someweather}
```

```
185 \def\funcforalign{%
186   \ifcase\nr\relax
187     \def\mp@align{\centering}%
188   \or\def\mp@align{\flushright}%
189   \or\def\mp@align{\flushleft}%
190   \or\let\mp@align\relax
191   \fi
192 }
193 % Keys ‘color’ and ‘mybool’ have no macro.
194 % Submitted value of key ‘angle’ is ##1 → \mp@angle.
195 \define@keylist{%
196   ord,color,gray!25,;
197   cmd,angle,45,\def\anglevalue{##1};
198   cmd,scale,1,\def\mydo####1{\do ####1};
199   choice,align,center,\funcforalign;
200   \listbreak;
201   bool,mybool,true,;
202   choice,weather,sunny,\edef\Weather{\val}%
203 }
204 \setkeys[KV]{fam}{angle=45,scale=1cm,weather=cloudy}
```

Note the number of parameter characters in the definition of \mydo. We will return to this matter in Subsection 7.2.1. 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 numerical order of 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 \funcforalign before hand, we can submit its replacement text directly to the macro \define@keylist, but, because \funcforalign contains a conditional, some care is needed in doing so (see Subsection 7.4). Once the key align has been defined, the macro \funcforalign can't be reused before the key align is set. This is because it is at key setting time that the function \funcforalign would be called. This is uneconomical: it is thus desirable to submit the key macro directly to \define@keylist irrespective of the presence of conditionals.

Please note the \listbreak token inserted on macro line 200 above. Because of it, the keys mybool and weather will not be read and defined; all the keys before \listbreak (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

```
205 \CKVS*{align}{center,right,left,justified}
```

```
\CKVS{election}{state,federal,congress,senate}

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

\def\funcfortextheight{%
  \ifx\@empty\mp@textheight
    \wlog{‘textheight’ value empty}%
  \else
    \wlog{‘textheight’ value not empty}%
  \fi
}

\def\funcforpaperwidth{\wlog{‘paperwidth’ was defined as %
  ordinary key.}}

\def\funcforalign{%
  \ifcase\nr\relax
    \def\mp@align{\centering}%
  \or\def\mp@align{\flushright}%
  \or\def\mp@align{\flushleft}%
  \or\let\mp@align\relax
  \fi
}
% ‘boolvar’, ‘paperheight’ and ‘evensidemargin’ have no
% key macros:
\define@keylist{%
  bool,boolvar,true,;
  ord,paperheight,\paperheight,;
  ord,paperwidth,\paperwidth,\funcforpaperwidth;
  cmd,textheight,\textheight,\funcfortextheight;
  cmd,textwidth,\textwidth,\funcfortextwidth;
  ord,evensidemargin,\evensidemargin,;
  tog,togvar,true,\iftogon{mp@togvar}{\def\catch####1{####1}}%
    {\def\gobble####1{}};
  choice,align,center,\funcforalign;
  choice,election,congress,\def\electiontype{##1};
  cmd,testdim,2cm,\long\def\funcfortestdim####1{%
    A test dimension ####1 \endgraf\bigskip}%
}

\setkeys[KV]{fam}{togvar=true,testdim=1cm,election=senate}
```

The macro `\electiontype` corresponds to `\val` for choice key `election`. Again, the intermediate/utility key macros can be reused only after their associated keys have been set.

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

Example

```
\def\keylistvector{%
  bool,boolvar,true,;
  ord,paperheight,\paperheight,;
  ord,paperwidth,\paperwidth,\funcforpaperwidth;
  cmd,textheight,\textheight,\funcfortextheight;
  cmd,textwidth,\textwidth,\funcfortextwidth;
  ord,evensidemargin,\evensidemargin,;
  tog,togvar,true,\iftogon{mp@togvar}{\def\catch####1{####1}}%
    {\def\gobble####1{}};
  choice,align,center,\funcforalign;
  choice,election,congress,\def\electiontype{##1};
  cmd,testdim,2cm,\long\def\funcfortestdim####1{%
    A test dimension ####1 \endgraf\bigskip}%
}
\define@keylist*\keylistvector
```

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 \KV@fam@testdim holds the macros:

Example

```
\def\mp@testdim{#1}
\long\def\funcfortestdim##1{A test dimension ##1},
```

where ⟨#1⟩ is the value (3cm) submitted for the key testdim. The number of parameter characters normally increases in the macro \define@keylist (see Subsection 7.2.1). After setting the keys, you can do \show \mp@align and \show \KV@fam@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 \funcforpaperwidth, while \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 list registers. Some examples are provided below:

Example

```
\NewToks[temptoks]{a,b}
% See page 47 for definition of \NewToks and related commands.

\temptoksa{\ifmp@boola\def\do#1{%
  \def#1##1##2{\expandafter\expandafter\expandafter\in@
  \expandafter\expandafter\expandafter{\expandafter##1%
  \expandafter}\expandafter{##2}}}\fi}

\temptoksb{\iftogon{mp@toga}{\def\order#1{Use ‘#1’ now!}}%
  {\def\altorder#1{Don’t use ‘#1’ now!}}}

\define@keylist{3,boola,true,\the\temptoksa;
  4,toga,true,\the\temptoksb}

\setkeys[KV]{fam}{boola=true,toga=true}
```

The advantage of using token list registers is that the parameter characters need not be doubled in the token list registers, unlike when using macros. The token list register `\temptoksa` can be reused as soon as the key `boola` has been set. See Subsection 7.4.1 for more information on using macros and token list registers to parse key functions.

7.2.1 Parameterized macros in key macros

The examples in Subsection 7.2 would have provided some glimpse of the rules guiding the use of parameter characters in key macros. The general rules are as follows:

a) When key macros are parsed through token list registers, the parameter characters shouldn’t be doubled.
b) When key macros are parsed via intermediate macros, the parameter characters should be doubled but only once.
c) In all other cases (i.e., when using `\define@keylist` and its starred variant) the parameter characters should be doubled twice.

TEX will flag a fatal error when any of these rules is breached. The following examples illustrate the use of these rules. The commands `\skif`, `\skifx`, `\skelse` and `\skfi` are described in Subsection 7.4.2.

Example

```
\define@keylist{%
  tog,toga,true,\iftogon{mp@toga}{%
    \def\swear####1{Repeat after me: ‘####1’!}%
  }{%
    \let\swear\@gobble
  }%
}
\setkeys[KV]{fam}{toga=true}

\NewToks[temptoks]{a}
```

```
\temptoksa={\long\def\funcforproclaim#1%
  {A proclaimed statement: #1}}
\define@keylist{%
  bool,boola,true,\skif{mp@boola}\def\yes####1%
    {Accept `####1'!}\skelse\def\no####1{Reject `####1'!}\skfi;
  cmd,proclaim,Statement,\the\temptoksa
}
\setkeys[KV]{fam}{boola=true,proclaim=nature}

\CKVS{align}{left,right,center}

\define@keylist{choice,align,center,
  \skifcase\nr
    \def\hold####1{\def####1########1{===########1===}}%
  \skor
    \def\hold####1{\def####1########1{+++########1+++}}%
  \skfi
}
\setkeys[KV]{fam}{align=right}

\CKVS{focus}{left,right,center}
\def\keylistvector{%
  choice,focus,center,\def\hold####1%
  {\def####1########1{===########1===}}%
}
\define@keylist*\keylistvector
\setkeys[KV]{fam}{focus=right}

\def\keylistvector{cmd,keya,xxx,\def\hold####1%
  {\def####1########1{===########1===}}}
\define@keylist*\keylistvector
\setkeys[KV]{fam}{keya=yyy}

\def\funcforkeyb{\def\hold##1{\def##1####1{===####1===}}}
\define@keylist{cmd,keyb,xxx,\funcforkeyb}
\setkeys[KV]{fam}{keyb=yyy}

\define@cmdkey[KV]{fam}[mp@]{keyc}[xxx]{\def\hold##1{##1}}
\setkeys[KV]{fam}{keyc=yyy}

\def\keylistvector{%
  ord,keyda,aaa,\def\hold####1%
    {\def####1########1{===########1===}};
  cmd,keydb,bbb,\def\althold####1%
    {\def####1########1{***########1***}}%
}
\define@keylist*\keylistvector
\setkeys[KV]{fam}{keyda=xxx,keydb=yyy}
```

```
% The next one fails. Why?
\define@keylist{ord,keye,unknown,\def\hold##1%
  {\def##1####1{####1}}}

\define@keylist{%
  ord,keyfa,xxx,
  \skifx\x\y
    \def\hold####1{\def####1########1{===########1===}}%
  \skelse
    \def\hold####1{\def####1########1{***#########1***}}%
  \skfi;
  cmd,keyfb,yyy,
  \SKV@ifx\x\y{%
    \def\nosupergobble####1{\def####1########1{########1}}%
  }{%
    \def\supergobble####1{\def####1########1{}}%
  }%
}
\setkeys[KV]{fam}{keyfa=value,keyfb=value}

\def\funcforboolb{\ifmp@boolba\def\do##1{%
  \def##1####1####2{\expandafter\expandafter\expandafter\in@
  \expandafter\expandafter\expandafter{\expandafter####1%
  \expandafter}\expandafter{####2}}}\fi}

\define@keylist{3,boolba,true,\funcforboolb;3,boolbb,true,;}
\setkeys[KV]{fam}{boolba=true}

\NewToks[temptoks]{a,b}

\temptoksa{\ifmp@boolc\def\do#1{%
  \def#1##1##2{\expandafter\expandafter\expandafter\in@
  \expandafter\expandafter\expandafter{\expandafter##1%
  \expandafter}\expandafter{##2}}}\fi}
\temptoksb{\iftogon{mp@togb}{\def\tempa#1{Use #1}}%
  {\def\tempb#1{Don't use #1}}}
\define@keylist{3,boolc,true,\the\temptoksa;
  4,togb,true,\the\temptoksb}

\setkeys[KV]{fam}{boolc=true,togb=true}
\do\x \def\y{x} \def\z{xxx} \x\y\z

\def\keylistvector{bool,boold,true,
  \ifswitchon{mp@boold}{%
    \def\hold####1{\def####1########1{***#########1***}}%
  }{%
    \def\hold####1{\def####1########1{===########1===}}%
  }%
}
\define@keylist*\keylistvector
```

```
367 \setkeys[KV]{fam}{boold=true}
```

7.3 Input error

Native-boolean, toggle-boolean 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 native-boolean and toggle keys are `true` and `false`. The valid values of choice keys are set by the user via `\CKVS`. The default input error is defined by the macro `\SKV@inputerr`. It takes two arguments (i.e., value and key) and can be customized by the user.

7.4 Conditionals in key macros

The TEX conditional primitives `\if`, `\ifx`, `\else` 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 loops used in the `skeyval` package to define keys by means of `\define@keylist`. There are many possible approaches to resolving this problem, but only four appear to be attractive (see Subsections 7.4.1 to 7.4.4).

7.4.1 Burying conditionals in intermediate macros or token list registers

Key macros/functions involving conditional operations such as

Example

```
368 \ifmp@bool \do \else \donot \fi
```

can be submitted to `\define@keylist` via intermediate macros, as seen above (in Subsection 7.2), but the approach isn't economical and thus not advisable. Nevertheless, we give more examples of deploying intermediate macros below. Let the key macro prefix be `mp@`, the key prefix be `KV`, and the key family be `fam`.

Suppose we want to submit the following:

Example

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

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

Example

```
370 \def\funcforbool{\ifmp@bool \do \else \donot \fi}
```

and then do

Example

```
371 \define@keylist{3,bool,true,\funcforbool},
```

which will execute \funcforbool when the key bool is set. One significant drawback of this approach is that once the key bool has been defined by the above statement, the function \funcforbool may not be redefined and reused before the key bool is set. This is wasteful and not advisable. This approach is included here only for demonstration purposes. The schemes in Subsections 7.4.2 to 7.4.3 are preferable.

As another example, we may do

Example

```
372 \def\funcforboola{\ifmp@boola\def\do##1{%
373   \def##1####1####2{\expandafter\expandafter\expandafter\in@
374   \expandafter\expandafter\expandafter{\expandafter####1%
375   \expandafter}\expandafter{####2}}}\fi}

376 \define@keylist{3,boola,true,\funcforboola}

377 \setkeys[KV]{fam}{boola=true}

378 \def\y{x} \def\z{xxx} \do\x \x\y\z
```

Token list registers can be used here economically instead of macros. Below we define one native-boolean key and one toggle-boolean key:

Example

```
379 \NewToks[temptoks]{a,b}

380 \temptoksa{\ifmp@boola\def\do#1{%
381   \def#1##1##2{\expandafter\expandafter\expandafter\in@
382   \expandafter\expandafter\expandafter{\expandafter##1%
383   \expandafter}\expandafter{##2}}}\fi}

384 \temptoksb{\iftogon{mp@toga}{\def\order#1{Use `#1' now!}}{}}

385 \define@keylist{3,boola,true,\the\temptoksa;
386   4,toga,true,\the\temptoksb}

387 \setkeys[KV]{fam}{boola=true,toga=true}
```

You can see the significant reduction in the number of parameter characters when using token list registers. The utility token list registers \temptoksa and \temptoksb can be reused to define many other keys as soon as the keys boola and toga have been set. However, as noted earlier, the approach of using intermediate macros and token list registers to parse arguments to \define@keylist is not attractive because of the overheads in the number of macros and token list registers.

7.4.2 Using pseudo-primitives to submit the conditionals

There are two downsides to the above approach of hiding conditionals in macros: the macros have to be defined and, although they can be redefined and reused

(after the associated key has been set), they tend to defeat the initial aim of the package, which is to economize on tokens.

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

Example

```
\ifmp@mybool
  \def\hold##1{\def##1####1{*******1***}}%
\else
  \def\hold##1{\def##1####1{===####1===}}%
\fi
```

where the key prefix KV, key family fam, and the macro prefix mp@ are assumed to have been defined previously. Then, instead of hiding the conditional in an intermediate macro, we may adopt the following:

Example

```
\define@keylist{3,mybool,true,
  \skif{mp@mybool}\def\hold##1{\def##1####1{*******1***}}%
  \skelse\def\hold##1{\def##1####1{===####1===}}\skfi
}

\setkeys[KV]{fam}{mybool=true or false}
\hold\x
```

Here we have used the pseudo-primitives \skif {mp@mybool}, \skifx, \skelse and \skfi for the commands \ifmp@mybool, \ifx, \else and \fi, respectively, to hide the latter four from TEX's scanning and skipping mechanism. It should be noted that \skif {mp@mybool} requires that the argument ⟨mp@mybool⟩ be enclosed in braces. Something like \skifmp@mybool will be interpreted by TEX as an undefined control sequence when the key mybool is being set. Defining the command \skif {mp@mybool} to be \ifmp@mybool before hand would have failed because TEX's scanner would then get the hint of the assignment.

Note: We haven't found any package that has defined \skif, \skifx, \skelse, \skfi, \skifcase or \skor. The xifthen package, for example, uses \OR, not \skor. If the situation changes in the future (i.e., if a package is observed to have defined any of these commands), they will be appropriately modified in the skeyval package. Information and feedback from package users is solicited in this regard.

We have redefined the \setkeys command of the xkeyval package to recognize that \skif {boolean}, \skifx, \skelse, \skfi, \skifcase and \skor stand for \ifboolean, \ifx, \else, \fi, \ifcase and \or, respectively. The redefined \setkeys command has the same syntaxes 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⟩}.
```

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 {}{}`.

Conditionals involving \ifcase: The case of conditionals involving `\ifcase` can be handled in the same way as those involving `\if`:

Example

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

\temptoksa{\ifcase\nr\relax
  \def\mp@focus{\centering}\or\def\mp@focus{\flushright}
  \or\def\mp@focus{\flushleft}\or\let\mp@focus\relax\fi}

\define@keylist{5,focus,center,\the\temptoksa}
```

This can be written more compactly as follows, which obviates the need for intermediate macros and list registers:

Example

```
\define@keylist{menu,focus,center,\skifcase\nr\relax
  \def\mp@focus{\centering}\skor\def\mp@focus{\flushright}
  \skor\def\mp@focus{\flushleft}\skor\let\mp@focus\relax\skfi}
```

Here, the skeyval package uses `\skifcase`, `\skor`, and `\skfi` for `\ifcase`, `\or` and `\fi`, respectively; otherwise, TEX would be grumpy.

The key `focus` can now be readily set: `\setkeys [KV]{fam}{focus=left}`.

7.4.3 Using switches to submit the conditionals

The approaches of Subsection 7.4.2 provide a familiar TEX-like syntax for submitting conditionals to `\define@keylist`. There is yet another approach that we developed. It is related to the native TEX boolean conditional. By *switches* we mean the usual TEX's `\iftrue` and `\iffalse` booleans, but expressed in different syntax and semantics.

A new switch can be introduced by using the following command:

Macro

```
\newswitch[⟨optional prefix⟩]{⟨boolean⟩}[⟨optional state⟩]
```

This provides a new native-boolean register `\if⟨prefix⟩⟨boolean⟩` if the register didn't already exist; if the register already existed, an error is flagged. The optional ⟨state⟩ can be either `true` (or `on`) or `false` (or `off`); the switch will be initialized to that state. With this definition, you can issue `\if⟨switch⟩`, `\⟨switch⟩true`, and `\⟨switch⟩false`. One advantage of `\newswitch` is that a switch can start off as either `true` (or `on`) or `false` (or `off`), unlike the classical

TEX's case in which all booleans start off as `false`. Also, switches can be used where primitive TEX conditionals may prove impossible. For example, we know that you can't do `\let \ifabc \iftrue` within the body of a conditional text without hiding the assignment from TEX's scanning mechanism.

Example

```
\newswitch{boola}  →  \newif\ifboola

\newswitch[bool]{b}[true]  →  \newif\ifboolb \boolbtrue

\newswitch[bool]{c}[on]  →  \newif\ifboolc \boolctrue

\newswitch[bool]{d}[off]  →  \newif\ifboold \booldfalse
```

You can define many switches in a row by the following command:

Macro

```
\NewSwitches[⟨optional prefix⟩]{⟨switches⟩}[⟨optional state⟩]
```

Each member of ⟨switches⟩ is prefixed with ⟨prefix⟩ upon definition. The optional ⟨state⟩ can be either `true` (or `on`) or `false` (or `off`).

Here are some examples:

Example

```
\NewSwitches{x,y,z}[off]
    →    \newswitch{x}{false} \newswitch{y}{false}
         \newswitch{z}{false}

\NewSwitches[skv@]{u,w}[true]
    →    \newswitch{skv@u}{true} \newswitch{skv@w}{true}
```

Switches may be set and tested using the following commands:

Macro

```
\setswitch{⟨switch⟩}{⟨value/state⟩}

\switchon{⟨switch⟩}      →    \⟨switch⟩true

\switchtrue{⟨switch⟩}    →    \⟨switch⟩true

\switchoff{⟨switch⟩}     →    \⟨switch⟩false

\switchfalse{⟨switch⟩}   →    \⟨switch⟩false

\ifswitchon{⟨switch⟩}{⟨true text⟩}{⟨false text⟩}

\ifswitchtrue{⟨switch⟩}{⟨true text⟩}{⟨false text⟩}

\ifswitchoff{⟨switch⟩}{⟨not true⟩}{⟨not false⟩}
```

```
\ifswitchfalse{⟨switch⟩}{⟨not true⟩}{⟨not false⟩}
```

Example

```
\NewSwitches{w,x,y,z}

\setswitch{w}{true}   →   \setswitch{w}{on}
\setswitch{x}{on}     →   \setswitch{x}{true}
\setswitch{y}{false}  →   \setswitch{y}{off}
\setswitch{z}{off}    →   \setswitch{z}{false}

\ifswitchon{x}{\def\xx{On}}{\def\xx{Off}}

\ifswitchoff{y}{\def\yy{Off}}{\def\yy{On}}
```

Macro

```
\defswitch[⟨optional prefix⟩]{⟨boolean⟩}[⟨optional state⟩]
```

This is similar to \newswitch but raises no errors even if the register ⟨boolean⟩ already exists.

Now how do we use switches in \define@keylist? Suppose the key prefix is KV, the key family is fam, and the key macro is mp@. Then the following works:

Example

```
\define@keylist{3,swa,true,
  \ifswitchon{mp@swa}{\def\say{Swtich ‘a’ is true}}
    {\def\say{Swtich ‘a’ is false}}}

\setkeys[KV]{fam}{swa=true or false or on or off}
```

Note: Please note that “switch keys” are actually native-boolean keys (bool or type 3 key). When setting native-boolean and toggle-boolean keys you can supply either true, false, on or off as values. We have modified the relevant internal macros of the xkeyval package to make this possible for the native-boolean keys as well.

7.4.4 Using toggles to submit the conditionals

Toggle booleans, described in Section 5, can also be used to circumvent the problem of matching \if and \fi in difficult circumstances, since toggles aren't TEX primitives, and, as noted in Section 5, toggles are very economical. For example, the following works:

Example

```
\define@keylist{4,toga,true,
  \iftogon{mp@toga}{\def\say{Toggle ‘a’ is true}}%
    {\def\say{Toggle ‘a’ is false}}}
```

```
446 \setkeys[KV]{fam}{toga= true or false}
```

where the key prefix, key family, and macro prefix are still assumed to be KV, fam, mp@, respectively. Recall that toggle keys are type 4 keys.

8 Key command and key environment

The \define@keylist macro of Section 7, together with the \newkeylist macro of Subsection 9.2.1, provides an attractive toolkit for developing a system for defining commands using keys, in the manner of the keycommand package but with a more compact user interface. See the skeycommand package for details.

9 Checking and redefining keys

9.1 Checking the status of a key

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

Macro

```
447 \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

```
448 \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 Subsection 10.3.

Macro

```
449 \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 Subsection 10.4.

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 these commands before defining every key can be laborious.

Consider the following two scenarios:

Example

```
450 \define@key[KV]{fam}{keya}[$\star$]{\def\tempa##1{##1}}

451 \define@boolkey[KV]{fam}{keya}[true]{%
452   \ifKV@fam@keya\def\tempb{#1}\fi}

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

Obviously the two definitions of ⟨keya⟩ are valid and will be implemented but the \setkeys command here 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 Subsection 5.2, 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. In fact, the problem can manifest itself in more ways than the scenario just depicted.

If the package option verbose is enabled, the skeyval package provides a warning system (by making an entry in the transcript log file) if an existing key is being redefined (within the same family and with the same prefix) by any of the following commands:

Macro

```
454 \define@key, \define@cmdkey, \define@cmdkeys,
455 \define@choicekey, \define@boolkey, \define@boolkeys,
456 \define@biboolkeys, \define@uniboolkeys, \define@biuniboolkeys,
457 \define@togkey, \define@togkeys, \define@bitogkeys,
458 \define@unitogkeys, \define@biunitogkeys
```

The machinery of Subsection 9.2.1 can be used to avoid inadvertently redefining existing keys.

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@boolkeys:
\newboolkeys[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨keys⟩}[⟨default⟩]

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

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

% Counterpart of \define@biuniboolkeys:
\newbiuniboolkeys[⟨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@togkeys:
\newtogkeys[⟨prefix⟩]{⟨family⟩}[⟨mp⟩]{⟨keys⟩}[⟨default⟩]

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

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

% Counterpart of \define@biunitogkeys:
\newbiunitogkeys[⟨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}.
```

The following aliases are defined in the skeyval package:

Macro

```
\define@menukey*+    →    \define@choicekey*+
\newmenukey*+        →    \newchoicekey*+
```

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 may need to be protected. Commands of the type \define@xxxkey are originally, in the xkeyval package, non-robust and remain so in the skeyval package.

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

```
525 \newordkey[KV]{fam}{keya}[$\star$]{\def\tempa##1{##1}}

526 \newboolkey[KV]{fam}{keya}[true]{%
527   \ifKV@fam@keya\def\tempb{#1}\fi}
```

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

Besides macros for defining keys, the skeyval package also introduces mechanisms for disabling, localizing, 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

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

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 with prefix in ⟨prefixes⟩, an informational (not error) message is logged in respect of this member, but only if the package option verbose is selected.

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

Macro

```
529 \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

```
\def\SKV@disabledkey@err{%
  \PackageError{skeyval}{%
    Key ⟨key⟩ with prefix ⟨prefix⟩ in family ⟨family⟩
    was disabled on input line ⟨lineno⟩
  }{%
    You can't set or reset ⟨key⟩ at this
    late stage. Perhaps you're required to set it
    earlier, within a package or in the document's preamble.
  }%
}
```

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 (if the package option verbose is selected), since the situation isn't fatal to the outcome of the LaTeX pass or the document.

Disabled keys can be redefined with commands in the \define@xxxkey category but not with commands in the \newxxxkey 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 Subsections 10.3 to 10.4).

10.2 Localizing keys

By localizing a key we mean disabling a key at the end of the current class or package file. This is basically the command \disable@keys executed on the hook \SKV@BeforeClassEnd or \SKV@BeforePackageEnd, depending on \@currext. The hooks \SKV@BeforeClassEnd and \SKV@BeforePackageEnd are described in Section 14 (macro lines 857 to 858).

The syntax for localizing keys is exactly like that for disabling keys:

Macro

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

If any of the members of the set ⟨keys⟩ is not found in any of the members of ⟨families⟩ and with a prefix from ⟨prefixes⟩, an informational message is written into the log file (if the package option verbose is selected), but no errors are flagged.

The starred variant of \localize@keys disables the keys listed in ⟨keys⟩, not at the end of the package or class file, but right before the start of document (i.e., at the boundary between the document preamble and \AtBeginDocument):

Macro

541 \localize@keys*[⟨prefixes⟩]{⟨families⟩}{⟨keys⟩}.

The hook used here is \SKV@BeforeDocumentStart, described in Section 14, macro line 862.

A key can be localized as soon as it is defined. In fact, a key can be localized even before it is defined: the actual disablement of the key will take place at the execution of the contents of \SKV@BeforeClassEnd or \SKV@BeforePackageEnd or \SKV@BeforeDocumentStart.

Localized keys can be redefined with commands in the \define@xxxkey category, but not with commands in the \newxxxkey category.

10.3 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

542 \ReserveKeyPrefixNames{⟨list⟩}

This allows the package developer to bar the future use of names appearing in ⟨list⟩ as key prefixes when defining, disabling, reserving and suspending keys; but not when setting keys. The ⟨list⟩, whose members are comma-separated, can be populated by the package developer as required.

Macro

543 \ReserveMacroPrefixNames{⟨list⟩}

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

Macro

544 \ReserveFamilyNames{⟨list⟩}

This reserves family names ⟨list⟩ and prevents further use of members of ⟨list⟩.

Note: The lists in these macros are scanned only when defining, disabling, reserving or suspending keys, and not when setting existing keys. If the lists were also to be scanned when keys are being set, then a situation could arise in which existing keys (including those defined by prior packages) couldn't be set.

These macros could be used, for instances, to secure against future use the key prefixes, macro prefixes, and key families that have used in a new style or class file. For example, we have used these facilities to bar users of the skeyval

package from using the key prefix SKV, the family name skeyval, and the macro prefix SKV@ to define new keys in their packages.

Macro

```
\ReserveKeyPrefixNames*{⟨list⟩}
\ReserveMacroPrefixNames*{⟨list⟩}
\ReserveFamilyNames*{⟨list⟩}
```

These starred variants take effect at end of current package; the unstarred ones above assume immediate effect. If the developer wants to use any member of ⟨list⟩ in his own package, it may be necessary for him to use the starred versions.

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

Macro

```
\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

```
\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 (if the package option verbose is selected) and ignore that key. Defined keys and suspended keys can also be unreserved, which is equivalent to removing the keys (see Subsection 10.5).

10.4 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

```
\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

551 `\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.

10.5 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

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

Removed keys can't be restored but can be redefined with the commands in both the \newxxxkey and \define@xxxkey categories.

11 User-value keys

We define these keys as those for which the user must supply values at key setting time whether or not the keys have default values. All the commands for defining new keys have a facility for providing the default value of a key, which would be used by the \setkeys macro if the user didn't supply a value for the key. If no default value has been specified for a key at definition time and no value is provided at key setting time, the xkeyval package will issue a fatal error. This scenario is preserved by the skeyval package. In addition, the skeyval package introduces a facility for requiring a user to supply a value for a key whether or not that key had a default value at definition time. Why is this necessary or useful? You may specify default values for keys in a package or class file to aid future revisions of the package, or for other purposes, but such values may not be suitable for all users—or indeed for any user. Examples of this type of situation abound: the signatory to a letter, the module code or title in a faculty programme, etc.

The following command can be used to require a user to supply a value for a key at key setting time, whether or not that key has a default value:

Macro

553 `\uservaluekeys[⟨prefix⟩]{⟨family⟩}{⟨keys⟩}`

where ⟨keys⟩ is the list of keys with ⟨prefix⟩ and in ⟨family⟩ for which the user must supply values at key setting time. It is obviously not logical to loop over key prefixes or families in this case. What the command \uservaluekeys

does is to populate the container \SKV@⟨prefix⟩@⟨family⟩@uservalue which is scanned for user-value key names at key setting time.

The \uservaluekeys macro works incrementally, i.e., new inputs are added to an existing list for the family in question only if they haven't previously been included.

Example

```
554 \newcmdkey[KV]{fam}[mp@]{keya}[12pt]{\def\x{#1}}
555 \newboolkey[KV]{fam}[mp@]{boola}[true]%
556   {\ifmp@boola\def\x{#1}\fi}
557 \uservaluekeys[KV]{fam}{keya,boola}
558 \setkeys[KV]{fam}{keya,boola=true}
559         → Error: no value supplied for ‘keya’
```

11.1 Using pointers to dynamically indicate user-value keys

Instead of using the macro \uservaluekeys to accumulate user-value keys, there is another way to dynamically specify these keys at key definition time: by using pointers. At key definition time, the pointers \uservalue and \guservalue can be associated with a user-value key. In the following statements, the pointer \uservalue specifies that the user of the affected key must supply a value at the time of using/setting the key. The pointer \uservalue has local effect, i.e., its impact can't escape local groups; on the other hand, \guservalue has global effect, i.e., using it within or out of a local group means that the user of the affected key must specify a value for the key at key setting time. \guservalue ensures that the internal container \SKV@⟨prefix⟩@⟨family⟩@uservalue is defined globally so that the settings can escape local groups.

Example

```
560 \newordkey[KV]{fam}{\uservalue{keya}}[12pt]{\def\x{#1}}
561 \define@togkey+[KV]{fam}[mp@]{\guservalue{toga}}[true]%
562 {%
563   \iftogon{mp@toga}{\def\x{#1}}{}%
564 }{%
565   \@latex@error{Value ‘#1’ not valid}\@ehc
566 }
```

As these examples show, the newly introduced pointers (namely, \uservalue and \guservalue) can be used to dynamically build a list of user-value keys. See Section 12 for more comments on pointer systems.

12 Extensions to the pointer system of the xkeyval package

The xkeyval package introduced a key pointer system. This basically involves the pointers \savevalue, \gsavevalue, and \usevalue. However, by the mech-

anism of that package, these pointers could be used only within the \setkeys command or context. In the key definition commands, the \usevalue pointer could also be used in default values of keys, as in

Example

```
\define@key{fam}{keya}{\def\keya{#1}}
\define@key{fam}{keyb}[\usevalue{keya}]{\def\keyb{#1}}
\define@key{fam}{keyc}[\usevalue{keyb}]{\def\keyc{#1}}

\setkeys{fam}{\savevalue{keya}=test}
\setkeys{fam}{\savevalue{keyb}}% Yes, this also works.
\setkeys{fam}{keyc}
```

The default values of keys are called (invoked) within \setkeys. The pointers can't, however, be used as part of key names outside default values. The following, e.g., fails:

Example

```
\define@key{fam}{\savevalue{keya}}{\def\keya{#1}}
```

The skeyval package has extended the pointer system to be issuable as part of key names within key definition commands in the two scenarios illustrated above (within and outside default values). Moreover, the pointers \savevalue and \gsavevalue can be deployed concurrently with \uservalue and \guservalue of Subsection 11.1 within the same key definition command. In combining the two pointer subclasses (i.e., \savevalue subclass and \uservalue subclass) in the same key definition command, it doesn't matter which subclass comes first, as the following examples show.

The pointers \savevalue and \gsavevalue make entries into the container \XKV@⟨prefix⟩@⟨family⟩@save that is used by the xkeyval package to hold keys whose values should be saved at key setting time. The difference between \savevalue and \gsavevalue is that the former has a local effect while the latter can escape local groups (similar to the group properties of \uservalue and \guservalue of Subsection 11.1). The pointers \savevalue and \gsavevalue of the skeyval package are entirely compatible with those of the xkeyval package. One additional new feature is that the pointer \gsavevalue prompts the global revision of the container \XKV@⟨prefix⟩@⟨family⟩@save and also makes global pointer entries* of the affected keys into the container. The effects of the new feature are illustrated by the following examples. Depending on application, this new feature may be more attractive than the traditional one implemented via \setkeys. One obvious advantage of the new system emanates from the fact that some keys do not have default values.

*By *global pointer entry* we mean an entry like \global {keya} for keya into the container \XKV@⟨prefix⟩@⟨family⟩@save.

12.1 Examples

12.1.1 Legacy xkeyval pointer features

The following provide examples of legacy pointer features of the xkeyval package (key pointers at key setting time):

Example

```
574 \savekeys[KV]{fam}{keya,\global{keyb}}
575 \gsavekeys[KV]{fam}{keyc,keyd,\global{keye}}

576 \setkeys[KV]{fam}{\gsavevalue{keyd}=yyy,
577   keye=\usevalue{keyd}}
```

If we had included \global {keyb} in \gsavekeys of macro line 575, its entry in \savekeys would have been overwritten, since keys in the container \XKV@KV@fam@save normally get overwritten if they have the same name. The macro \gsavekeys ensures the global definition of \XKV@KV@fam@save when the keys keyc, keyd and keye are being included, while \global {keyb} ensures that, when keyb is used in a \setkeys command, its value will be saved globally to \KV@fam@keyb@value. When the keys keyc and keyd are set, their values will be saved locally, even though the container \XKV@KV@fam@save was defined globally when the keys keyc and keyd were inserted. However, when keys keyb and keye are set, their values will be saved globally (even though keyb appears in \savekeys and not in \gsavekeys).

The pointer \gsavevalue {keyd} of macro line 576 will ensure that the value of keyd is saved globally to \KV@fam@keyd@value at \setkeys.

12.1.2 Extensions by skeyval package

The following provide examples of new pointer features enabled by the skeyval package (key pointers at key definition time):

Example

```
578 \define@key[KV]{fam}{\savevalue{keya}}[xxx]{\def\x{*#1*}}

579 \newordkey[KV]{fam}{\gsavevalue{keyb}}[zzz]{\def\x{=#1=}}

580 \newtogkey+[KV]{fam}[mp@]{\savevalue\uservalue{toga}}[true]%
581 {
582   \iftogon{mp@toga}{\def\x{#1}}{}%
583 }{%
584   \@latex@error{Value `#1' not valid}\@ehc
585 }

586 \define@cmdkey[KV]{fam}[mp@]{\uservalue\savevalue{keyc}}%
587   [www]{\def\x{#1}}

588 \newboolkey+[KV]{fam}[mp@]{\gsavevalue\uservalue{boola}}%
589 [true]{%
```

```
590   \ifmp@boola\def\x{#1}\fi
591 }{%
592   \@latex@error{Value `#1' not valid}\@ehc
593 }

594 \define@cmdkey[KV]{fam}[mp@]{\guservalue\savevalue{keyd}}%
595   [www]{\def\x{#1}}

596 \newchoicekey*+[KV]{fam}{\guservalue\gsavevalue{align}}%
597 [\val\nr]{center,right,left}[center]%
598 {%
599   \ifcase\nr\relax
600     \def\@align{\centering}%
601   \or
602     \def\@align{\flushright}%
603   \or
604     \def\@align{\flushleft}%
605   \fi
606 }{%
607   \@latex@error{Inadmissible value `#1' for align}\@ehc
608 }

609 \setkeys[KV]{fam}{keya=Hello World,keyb=\usevalue{keya}}
```

With the new mechanism of the skeyval package, the \gsavevalue pointer in the command on macro line 579 will ensure that \global {keyb} (not keyb) is inserted in the container \XKV@KV@fam@save and that this container is updated globally after \global {keyb} has been inserted. At \setkeys, in view of the entry \global {keyb}, the value of keyb will be saved globally. The same applies to keys boola and align. This thus has a double effect. Keys keyc and keyd will be saved locally.

13 Setting keys: list normalization

We have redefined the \setkeys command of the xkeyval package in two respects: firstly to accommodate the use of the \skif, \skifx, \skelse, and \skfi macros of Subsection 7.4.2, and secondly to automatically convert double (or even multiple) commas 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 and which we wish to address:

Example

```
610 \define@key[KV]{fam}{width}[1cm]{}
611 \define@key[KV]{fam}{color}[black]{}
612 \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.

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

```
613 \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.

14 Miscellaneous macros

This package is predominantly about LaTeX keys and their efficient creation and management, but it also contains many commands for general TeX programming, 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. The index provides a comprehensive quick resource locator for the commands.

Defining new commands

The following are provided in the skeyval package but you're advised to use the \TestProvidedCommand macro (described below) to test that you are really using the \newdef of the skeyval package:

Macro

```
614 \SKV@newdef*⟨cs⟩⟨parameters⟩{⟨replacement text⟩}
615 \newdef*⟨cs⟩⟨parameters⟩{⟨replacement text⟩}
```

These commands adopt TeX's syntax and accept parameter delimiters. They are both robust. The unstarred variant produces long macros. The command \newdef is defined in the skeyval package only if it hasn't been defined by a previously loaded package; the command \SKV@newdef, on the other hand, is always available. If ⟨cs⟩ was previously defined, both \SKV@newdef and \newdef will issue an error.

Defining robust commands

Macro

```
\SKV@robustdef*⟨cs⟩⟨parameters⟩{⟨replacement text⟩}
\robustdef*⟨cs⟩⟨parameters⟩{⟨replacement text⟩}
```

These use ε-TEX's \protected prefix to provide something resembling LATEX's \DeclareRobustCommand whilst conforming to TEX's \def interface. The unstarred variants produce long macros. These commands accept parameter delimiters and are all robust. The command \robustdef is defined in the skeyval package only if it hasn't been defined by a previously loaded package; the command \SKV@robustdef, on the other hand, is always available. If ⟨cs⟩ was previously defined, both \SKV@robustdef and \robustdef will issue an error. You can use the above \TestProvidedCommand to check whether or not you are using the \robustdef of the skeyval package.

TEX-like \providecommand

Macro

```
\SKV@providedef*⟨cs⟩⟨parameters⟩{⟨replacement text⟩}
\providedef*⟨cs⟩⟨parameters⟩{⟨replacement text⟩}
\SKV@providerobustdef*⟨cs⟩⟨parameters⟩{⟨replacement text⟩}
\providerobustdef*⟨cs⟩⟨parameters⟩{⟨replacement text⟩}
```

These emulate LATEX's \providecommand, but they conform to TEX's \def interface. The unstarred variants produce long macros. These commands accept parameter delimiters and are all robust. The commands \providedef and \providerobustdef are defined in the skeyval package only if they haven't been defined by a previously loaded package; the commands \SKV@providedef and \SKV@providerobustdef, on the other hand, are always available. Macros defined by \SKV@providerobustdef and \providerobustdef are robust, while those defined by \SKV@providedef and \providedef are nonrobust. If ⟨cs⟩ was previously defined, all these commands will simply ignore the new definition and enter a message to this effect in the log file (if the package option verbose is selected).

\requirecmd

Macro

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

This is explained in Subsection 5.1. 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 (if the package option verbose is selected) and the new definition is aborted.

Testing *provided* commands

Macro

```
624 \TestProvidedCommand⟨cs⟩{⟨true text⟩}{⟨false text⟩}
```

This can be used to test whether or not one is using the \newdef (or any other "provided" command) of the skeyval package. Here ⟨cs⟩ is either \newdef or any "provided" command. In fact, if you define any command using the macro \SKV@providedef or \SKV@providerobustdef (see below), you can verify by \TestProvidedCommand whether or not the new definition is the one in effect.

Example

```
625 \SKV@providerobustdef*\newcmd{\newcommand}

626 \TestProvidedCommand\newcmd{%
627   \@latex@info{‘\string\newcmd’ is ‘\string\newcommand’}%
628 }{%
629   \@latex@error{‘\string\newcmd’ isn’t %
630     ‘\string\newcommand’}\@ehd
631 }
```

Example

```
632 \SKV@providedef*\declarecommand{\newcommand}
633 \TestProvidedCommand\declarecommand{}%
634   {\let\declarecommand\newcommand}
```

Declaring new unique variables collectively

New definable variables can be introduced in sets by the following commands.

Macro

```
635 \NewBooleans[⟨optional prefix⟩]{⟨boolean list⟩}[⟨optional state⟩]
```

This provides, for each member of the comma-separated list ⟨boolean list⟩, a new native-boolean register if the register didn't already exist, otherwise an error is flagged. Each member of ⟨boolean list⟩ is prefixed with ⟨prefix⟩ upon definition. The optional ⟨state⟩ can be either true, false, on or off. on is synonymous with true, whilst off is equivalent to false.

Example

```
636 \NewBooleans[bool]{a,b,c}[true]
637   → \newif\ifboola \newif\ifboolb \newif\ifboolc
638     \boolatrue \boolbtrue \boolctrue

639 \NewBooleans{boold}  →  \newif\ifboold
```

Notice that members of the list ⟨boolean list⟩ don't have \if in their names.

Macro

```
\NewTogs[⟨optional prefix⟩]{⟨tog list⟩}[⟨optional state⟩]
```

This provides a new toggle register for each member of the comma-separated list ⟨tog list⟩ if the register didn't already exist, otherwise an error is flagged. Each member of ⟨tog list⟩ is prefixed with ⟨prefix⟩ upon definition. The optional ⟨state⟩ can be either true, false, on or off. on is synonymous with true, whilst off is equivalent to false.

Example

```
\NewTogs[tog]{a,b,c}[true]
  → \newtog{toga} \newtog{togb} \newtog{togc}
    \togon{toga} \togon{togb} \togon{togc}

\NewTogs{togd}  →  \newtog{togd}
```

Macro

```
\NewToks[⟨optional prefix⟩]{⟨toks list⟩}
```

This provides a new token list register for each member of the comma-separated list ⟨toks list⟩ if the register didn't already exist, otherwise an error is flagged. Each member of ⟨toks list⟩ is prefixed with ⟨prefix⟩ upon definition.

Example

```
\NewToks[toks]{a,b,c}
  → \newtoks\toksa \newtoks\toksb \newtoks\toksc

\NewToks{toksd}  →  \newtoks\toksd
```

Macro

```
\NewCounts[⟨optional prefix⟩]{⟨counter list⟩}
```

This provides a new counter register for each member of the comma-separated list ⟨counter list⟩ if the register didn't already exist, otherwise an error is flagged. Each member of ⟨counter list⟩ is prefixed with ⟨prefix⟩ upon definition.

Macro

```
\NewDimens[⟨optional prefix⟩]{⟨dimen list⟩}
```

This provides a new dimension register for each member of the comma-separated list ⟨dimen list⟩ if the register didn't already exist, otherwise an error is flagged. Each member of ⟨dimen list⟩ is prefixed with ⟨prefix⟩ upon definition.

Macro

```
\NewBoxes[⟨optional prefix⟩]{⟨box list⟩}
```

This allocates a new box register for each member of the comma-separated list ⟨box list⟩ if the box register didn't already exist, otherwise an error is flagged. Each member of ⟨box list⟩ is prefixed with ⟨prefix⟩ upon definition.

Example

```
652 \NewBoxes[box]{a,b,c}
653   → \newbox\boxa \newbox\boxb \newbox\boxc

654 \NewBoxes{boxd} → \newbox\boxd
```

Macro

```
655 \NewWrites[⟨optional prefix⟩]{⟨stream list⟩}
```

This allocates a new output stream for each member of the comma-separated list ⟨stream list⟩ if the stream didn't already exist, otherwise an error is flagged. Each member of ⟨stream list⟩ is prefixed with ⟨prefix⟩ upon definition.

Example

```
656 \NewWrites[write]{a,b,c}
657   → \newwrite\writea \newwrite\writeb \newwrite\writec

658 \NewWrites{writed} → \newwrite\writed
```

Macro

```
659 \NewReads[⟨optional prefix⟩]{⟨stream list⟩}
```

This allocates a new input stream for each member of the comma-separated list ⟨stream list⟩ if the stream didn't already exist, otherwise an error is flagged. Each member of ⟨stream list⟩ is prefixed with ⟨prefix⟩ upon definition.

The macros \NewBooleans, \NewToks, \NewCounts, \NewDimens, \NewBoxes, \NewWrites, and \NewReads are non-outer, unlike their primitive counterparts.

Defining new names

Macro

```
660 \SKV@csdef*+{⟨name⟩}{⟨definition⟩}
661 \SKV@csgdef*+{⟨name⟩}{⟨definition⟩}
662 \SKV@csedef*+{⟨name⟩}{⟨definition⟩}
663 \SKV@csxdef*+{⟨name⟩}{⟨definition⟩}
```

The variants of these commands without the plus (+) sign turn ⟨name⟩ into a control sequence in terms of ⟨definition⟩ whether or not the control was already defined. No error or warning messages are issued. The variants with plus (+) sign turn ⟨name⟩ into a control sequence if it wasn't already defined; if it is already defined, an error message is flagged. The starred (⋆) variants give "short" macros, while the unstarred variants yield "long" definitions. 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.*

Name use

Macro

```
664 \SKV@csuse{⟨name⟩}
```

This is similar to LATEX's legacy \@nameuse but returns \@empty (instead of an error) if ⟨name⟩ is undefined. This is due originally to etoolbox package.

\let assignments

Macro

```
665 \SKV@newlet{⟨cs1⟩}{⟨cs2⟩}
666 \NewLet{⟨cs1⟩}{⟨cs2⟩}
```

These assign ⟨cs2⟩ to ⟨cs1⟩ if ⟨cs2⟩ exists and if ⟨cs1⟩ isn't already defined, otherwise an error is flagged. The command \NewLet is defined in the skeyval package only if it hasn't been defined by a previously loaded package; the command \SKV@newlet, on the other hand, is always available. You can use \TestProvidedCommand (macro line 624) to test whether or not you are using the \NewLet command of the skeyval package.

Macro

```
667 \SKV@cslet{⟨name⟩}{⟨cs⟩}
668 \SKV@letcs{⟨cs⟩}{⟨name⟩}
669 \SKV@csletcs{⟨name⟩}{⟨name⟩}
```

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

Macro

```
670 \SKV@cslet*{⟨name⟩}{⟨cs⟩}
671 \SKV@letcs*{⟨cs⟩}{⟨name⟩}
672 \SKV@csletcs*{⟨name⟩}{⟨name⟩}
```

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

*The skeyval package contains other undocumented tools for defining new commands.

Number and dimension expressions

Macro

673 `\SKV@numdef+?{⟨num⟩}{⟨expression⟩}`

`\SKV@numdef` defines ⟨num⟩ from ⟨expression⟩ using ε-TEX's `\numexpr`. If ⟨num⟩ was previously undefined, it is first initialized with `\newcount` before the expression is built. If you do `\SKV@numdef \x {1+2+3}`, you would need to prefix `\x` with `\the` or `\number` in expressions. Expressions defined by `\SKV@numdef` can be used with TEX's operators such as `\advance` or `\multiply` and ε-TEX's `\numexpr` operator.

The plus sign (+) means that `\SKV@numdef` takes a control sequence name instead of a control sequence, while the question mark (?) implies that the macro `\SKV@numdef` effects a global assignment which can thus escape local groups.

Macro

674 `\SKV@dimdef+?{⟨dim⟩}{⟨expression⟩}`

`\SKV@dimdef` defines ⟨dim⟩ from ⟨expression⟩ using ε-TEX's `\dimexpr`. If ⟨dim⟩ was previously undefined, it is first initialized with `\newdimen` before the expression is built. If you do `\SKV@dimdef \x {1pt+2pt+3pt}`, you would need to prefix `\x` with `\the` in expressions. Expressions defined by `\SKV@dimdef` can be used with ε-TEX's `\dimexpr` operator.

The plus sign (+) means that `\SKV@dimdef` takes a control sequence name instead of a control sequence, while the question mark (?) implies that the macro `\SKV@dimdef` effects a global assignment which can thus escape local groups.

Verifying definability

Macro

675 `\SKV@ifdefinable⟨cs⟩{⟨function⟩}`
676 `\SKV@csifdefinable⟨cs name⟩{⟨function⟩}`

LATEX kernel'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. You can't do `\let \ifabc \iftrue` in such conditionals. The command `\SKV@ifdefinable`, which is robust, seeks to avoid these problems. `\SKV@csifdefinable` expects a control sequence name instead of a control sequence.

Macro

677 `\SKV@ifdefinable@n{⟨list⟩}`

The macro `\SKV@ifdefinable@n` accepts a comma-separated list of control sequence *names* whose definability are to be tested. It should be noted that the macro `\SKV@ifdefinable@n` doesn't accept ⟨function⟩, unlike the above

\SKV@ifdefinable. The aim of \SKV@ifdefinable@n is simply to test the definability of instances/members of ⟨list⟩.

Example

```
678 \SKV@ifdefinable@n{ax,ay,az}
```

Macro

```
679 \SKV@ifnewcmd*+[⟨optional parser⟩]{⟨list⟩}
```

The macro \SKV@ifnewcmd is similar to, but more versatile, than the command \SKV@ifdefinable@n. The star sign (⋆) in \SKV@ifnewcmd indicates that ⟨list⟩ is available in a macro, say \mylist; and the plus sign (+) shows \SKV@ifnewcmd that members of ⟨list⟩ (or \mylist) are control sequence names, otherwise they are control sequences (see examples below). The default value of the optional ⟨parser⟩ is "," (comma). The macro \SKV@ifnewcmd, like \SKV@ifdefinable@n but unlike \SKV@ifdefinable, doesn't execute any ⟨function⟩. Both \SKV@ifdefinable@n and \SKV@ifnewcmd are robust.

Example

```
680 \def\mylist{ax,ay,az}
681 \SKV@ifnewcmd*+[,]\mylist
682 \SKV@ifnewcmd*+\mylist
683 \SKV@ifnewcmd+{ax,ay,az}
684 \SKV@ifnewcmd{\ax\ay\az}
685 \def\my@list{\ax\ay\az}
686 \SKV@ifnewcmd*\my@list
687 \def\my@@list{ax;ay;az}
688 \SKV@ifnewcmd*+[;]\my@@list
689 \SKV@ifnewcmd\ax → \SKV@ifnewcmd+{ax} → \SKV@ifdefinable@n{ax}
```

Verifying the status of variables

Macro

```
690 \SKV@ifdef⟨cs⟩{⟨true⟩}{⟨false⟩}
691 \SKV@ifcsdef{⟨name⟩}{⟨true⟩}{⟨false⟩}
692 \SKV@ifundef⟨cs⟩{⟨true⟩}{⟨false⟩}
693 \SKV@ifcsundef{⟨name⟩}{⟨true⟩}{⟨false⟩}
```

These use ε-TEX's facilities to test the existence of the control sequence ⟨cs⟩ or control sequence name ⟨name⟩. These commands aren't robust and may be used to determine the current state of the macro replacement text, if such replacement text contains these commands. \relax'ed macros are considered undefined by *all* these commands. To test if a macro is \relax'ed, use the following commands:

Macro

```
694 \SKV@ifrelax⟨cs⟩{⟨true⟩}{⟨false⟩}
695 \SKV@ifcsrelax{⟨name⟩}{⟨true⟩}{⟨false⟩}
```

Macro

```
\SKV@ifdefax⟨cs⟩{⟨defined⟩}{⟨relaxed⟩}{⟨undefined⟩}
\SKV@ifcsdefax{⟨name⟩}{⟨defined⟩}{⟨relaxed⟩}{⟨undefined⟩}
```

These test if ⟨cs⟩ or ⟨name⟩ is defined, relaxed, or undefined. In using these three-valued logical tests, it is often easy to forget to include the null state (i.e., ⟨undefined⟩) because TEX is dominated by two-valued logical tests.

Undefining macros

Macro

```
\SKV@Undef*+?[⟨optional parser⟩]{⟨cs⟩}
```

This undefines the macros or control sequence names in the list ⟨cs⟩ (of nil, one or more elements) such that TEX will subsequently consider each element undefined. The star sign (*) indicates that ⟨cs⟩ is given as a macro whose *contents* are to be individually undefined, and the plus sign (+) shows that ⟨cs⟩ is made up of *control sequence names* instead of control sequences. The question mark (?) directs \SKV@Undef to *globally* undefine all the control sequences or names in ⟨cs⟩. Control sequence names are to be separated by the parser; control sequences shouldn't be separated. The default value of the ⟨parser⟩ is "," (comma). The command \SKV@Undef is robust (it will thus not expand in expansion contexts), but fragile arguments would need to be protected in expansion contexts.*

Example

```
\def\unwanted{tempa,tempb,tempc,temp1}
\SKV@Undef*+?[,]\unwanted
\def\unwanted{t1emp,te2mp,tem3p}
\SKV@Undef*+?[,]\unwanted
\SKV@Undef+?[;]{tempd;tempe;tempf}
\SKV@Undef\tempe
\SKV@Undef?{\tempea\tempeb\tempec\temped}
```

*Macros such as \@ifnextchar, \@ifstar, and those involving optional arguments normally can't be evaluated in expansion contexts. The same applies to the skeyval package macros with optional arguments. The etextools package introduced expandable variants of these commands, but in the contexts these commands are employed in the skeyval package, the expandable variants aren't particularly advantageous. The main reason is that some of our internal macros (e.g., looping macros) aren't amenable to full expansion anyway. Actually, the skeyval package provides the fully expandable variants (\SKV@TestOpt, \SKV@IfStar, \SKV@IfPlus, and \SKV@IfAsk) of the non-expandable commands \SKV@testopt, \SKV@ifstar, \SKV@ifplus, and \SKV@ifask. The commands \SKV@ifask and \SKV@IfAsk look for an optional question mark (?).

```
\SKV@Undef+{tempf}
\SKV@Undef?\tempg
\def\notwanted{\temph\tempi\tempj}
\SKV@Undef*\notwanted
```

The following non-generic variants avoid the above complications of signs, but they don't take lists:

Macro

```
\SKV@undef{⟨cs⟩} → \SKV@Undef{⟨cs⟩}
\SKV@gundef{⟨cs⟩} → \SKV@Undef?{⟨cs⟩}
\SKV@csundef{⟨cs name⟩} → \SKV@Undef+{⟨cs⟩}
\SKV@csgundef{⟨cs name⟩} → \SKV@Undef+?{⟨cs⟩}
```

Expansion control

Macro

```
\SKV@expox{⟨cs⟩}
```

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

Macro

```
\SKV@expcsox{⟨name⟩}
```

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

Macro

```
\SKV@exptx{⟨cs⟩}
```

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

Macro

```
\SKV@expcstx{⟨name⟩}
```

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

Macro

```
\SKV@expargs⟨n⟩⟨function⟩⟨arg1⟩⟨arg2⟩\@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@expargs, on the other hand, 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@expargs can be used to save \expandafter's, but caution should be exercised in deploying it: for example, the \edef it uses may expand too deeply in some cases. Also, precaution may be necessary when the expanded arguments (⟨arg1⟩ and/or ⟨arg2⟩) involve the TeX primitive \if. When invoking \SKV@expargs, the macros \SKV@expox and \SKV@exptx can be used to control the level of expansion.

Some trivial examples follow:

Example

```
719 \SKV@expargs{0}{\def\tempc#1#2}{\def\noexpand##1{##2}}\@nil
720 \tempc\tempa{aaa}
721 \tempc\tempb{abcaaabbccbca}
722 \SKV@expargs{2}\SKV@in@\tempa\tempb\@nil
723 \show\ifin@
```

These expressions show how \SKV@expargs can be used to economize on chains of \expandafter's. The expression on macro line 719, for example, isn't directly possible by \@expandtwoargs.

Checking values of choice keys

Choice keys should, by definition, have preordained values. This requirement can be useful even for non-choice keys, as we illustrate below.

Macro

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

This is an enhanced form of xkeyval package's \XKV@checkchoice. It checks if the user-submitted ⟨value⟩ of a key (say, ⟨keya⟩) is in the list ⟨altlist⟩. It executes ⟨true⟩ if ⟨value⟩ is found in ⟨altlist⟩ and ⟨false⟩ otherwise. Additionally, it returns \val for the expanded value of ⟨value⟩ and \nr for the numerical order of \val in the list ⟨altlist⟩. If ⟨value⟩ isn't found in ⟨altlist⟩, then \nr will return −1. If ⟨value⟩ and ⟨altlist⟩ are buried in macros, the macros are fully expanded before the search for ⟨value⟩ in the list

⟨altlist⟩ is effected. In that case, \val will hold the expanded form of ⟨value⟩ 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 ⟨value⟩ is given as \def \@value {center}. Obviously, \@value contains one of the elements of ⟨altlist⟩, but choice keys won't know this without the expansion of both \@altlist and \@value. This is *raison d'ê tre* of the \SKV@checkchoice macro.

Moreover, \SKV@checkchoice can be used in the definition of non-choice keys. In the following example we check the value of an *ordinary* key by means of \SKV@checkchoice:

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\@value{center}
\setkeys[KV]{fam}{keya=\@value}
```

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

Testing for substring

Macro

```
\SKV@in@{⟨substring⟩}{⟨string⟩}
```

This is similar to the LaTeX kernel's \in@{⟨substring⟩}{⟨string⟩} which tests if ⟨substring⟩ is in ⟨string⟩, but the present test avoids the problem of false result, which is typified by the following test:

Example

```
\in@{aa}{ababba}
```

This incorrectly returns \ifin@ as \iftrue. The macro \SKV@in@, on the other hand, correctly gives \ifin@ as \iffalse in this case. The command \SKV@in@ is robust.

Macro

```
743 \SKV@in@n{⟨substring⟩}{⟨string⟩}{⟨true text⟩}{⟨false text⟩}
```

This is similar to \SKV@in@ but, as shown here, the returned result has a different syntax.

Macro

```
744 \SKV@in@o{⟨substring⟩}{⟨string⟩}{⟨true text⟩}{⟨false text⟩}
```

This expands each of its two arguments once before the test.

Macro

```
745 \SKV@in@x{⟨substring⟩}{⟨string⟩}{⟨true text⟩}{⟨false text⟩}
```

This expands each of its two arguments fully before the test.

Macro

```
746 \in@tog{⟨substring⟩}{⟨string⟩}
```

In this case 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

```
747 \iftogon{in@}{⟨true text⟩}{⟨false text⟩}.
```

The command \in@tog is robust.

Macro

```
748 \in@tok{⟨substring⟩}{⟨string⟩}
```

Sometimes you want to use the LaTeX kernel's \in@{⟨substring⟩}{⟨string⟩} to test if ⟨substring⟩ is in ⟨string⟩ irrespective of their catcodes. The robust command \in@tok{⟨substring⟩}{⟨string⟩} 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@{⟨substring⟩}{⟨string⟩}. Actually, it calls \SKV@in@ to avoid false returns.

Testing equality of strings

Macro

```
749 \SKV@ifstrequal{⟨string1⟩}{⟨string2⟩}{⟨true⟩}{⟨false⟩}
750 \SKV@ifstrnotequal{⟨string1⟩}{⟨string2⟩}{⟨not true⟩}{⟨not false⟩}
```

```
751 \SKV@oifstrequal{⟨string1⟩}{⟨string2⟩}{⟨true⟩}{⟨false⟩}
752 \SKV@xifstrequal{⟨string1⟩}{⟨string2⟩}{⟨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 ⟨string1⟩ is equal (character code wise) to ⟨string2⟩, and ⟨false⟩ otherwise. Both ⟨string1⟩ and ⟨string2⟩ are detokenized before the test. The macro \SKV@oifstrequal is similar to \SKV@ifstrequal but first expands its arguments (the two strings ⟨string1⟩ and ⟨string2⟩) once before the test. The macro \SKV@xifstrequal first expands its arguments fully before the test.

Testing for empty or blank

Macro

```
753 \SKV@ifempty{⟨token⟩}{⟨true⟩}{⟨false⟩}
754 \SKV@ifnotempty{⟨token⟩}{⟨not true⟩}{⟨not false⟩}
755 \SKV@oifempty{⟨token⟩}{⟨true⟩}{⟨false⟩}
756 \SKV@xifempty{⟨token⟩}{⟨true⟩}{⟨false⟩}
```

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

Macro

```
757 \SKV@ifblank{⟨token⟩}{⟨true⟩}{⟨false⟩}
758 \SKV@ifnotblank{⟨token⟩}{⟨not true⟩}{⟨not false⟩}
759 \SKV@oifblank{⟨token⟩}{⟨true⟩}{⟨false⟩}
760 \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.

Verifying draft and final options

Macro

```
761 \SKV@ifdraft{⟨true⟩}{⟨false⟩}
762 \SKV@ifnotdraft{⟨not true⟩}{⟨not false⟩}
763 \SKV@iffinal{⟨true⟩}{⟨false⟩}
764 \SKV@ifnotfinal{⟨not true⟩}{⟨not false⟩}
765 \iftogon{draft}{⟨true⟩}{⟨false⟩}
766 \iftogoff{draft}{⟨not true⟩}{⟨not false⟩}
767 \iftogon{final}{⟨true⟩}{⟨false⟩}
768 \iftogoff{final}{⟨not true⟩}{⟨not false⟩}
```

These execute ⟨true⟩ or ⟨false⟩ depending on whether draft or final appears as true in the options list of \documentclass or \usepackage{skeyval}. The default is that final is true, which implies that draft is false by default. The keys draft and final are complementary native-boolean keys (see Section 4), which reduces the risk of mixing them. These commands are robust.

Macro

```
\ifdraft{⟨true⟩}{⟨false⟩}
\ifnotdraft{⟨not true⟩}{⟨not false⟩}
\iffinal{⟨true⟩}{⟨false⟩}
\ifnotfinal{⟨not true⟩}{⟨not false⟩}
```

These are also defined in the skeyval package, but because packages such as ifdraft package already exist, the skeyval package defines them only if they haven't already been defined. If they existed before skeyval package is loaded, they aren't redefined, and (if the package option verbose is selected) a warning is logged in the transcript file to indicate that the definition being used isn't from the skeyval package. The warning is logged only once. I am aware of the existence of only \ifdraft outside the skeyval package: therefore, the definition of \ifdraft is deferred until \AtBeginDocument. You can use \TestProvidedCommand to check the version of the \ifdraft that you are using.

Verifying dvi and pdf modes

Macro

```
\SKV@ifpdf{⟨true⟩}{⟨false⟩}
\SKV@ifnotpdf{⟨not true⟩}{⟨not false⟩}
\iftogon{pdf}{⟨true⟩}{⟨false⟩}
\iftogoff{pdf}{⟨not true⟩}{⟨not false⟩}
```

These execute ⟨true⟩ or ⟨false⟩ depending on whether dvi or pdf output is being produced. These commands are robust and may be used in expansion contexts.

Macro

```
\ifpdf{⟨true⟩}{⟨false⟩}
\ifnotpdf{⟨not true⟩}{⟨not false⟩}
```

These are available only if they haven't been previously defined by another package. If they existed outside the skeyval package, a warning is logged in the transcript log file (if the package option verbose is selected), but only once, and the commands are not redefined. In particular, the above \ifpdf is available only at \AtBeginDocument and is defined only if it doesn't already exist. The reason is that the popular hyperref package loads the ifpdf package and the ifpdf package will abort if it detects that \ifpdf has been defined by a package loaded earlier.

Note: The \ifpdf command of the ifpdf package is used in the \if ... \else ... \fi conditional, while the above \ifpdf command is used as indicated above. You can use the \TestProvidedCommand macro to check the version of the \ifpdf command that you are using.

Tests related to package loading

Macro

```
779 \@ifpackagecurrent{⟨package⟩}{⟨date⟩}{⟨true⟩}{⟨false⟩}
780 \@ifpackagenotcurrent{⟨package⟩}{⟨date⟩}{⟨not true⟩}{⟨not false⟩}
```

These executes ⟨true⟩ if the date of the current/loaded version ⟨package⟩ is greater than or equal to ⟨date⟩. This is similar to LaTeX's \@ifpackagelater but, unlike the latter, both \@ifpackagecurrent and \@ifpackagenotcurrent are robust. My main reason for these commands is that the nomenclature \@ifpackagelater is subject to the wrong interpretation of being space (rather than time) related.

Macro

```
781 \@afterpackageloaded{⟨package⟩}{⟨code⟩}
```

This executes ⟨code⟩ only after ⟨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, and to warn the user if ⟨package⟩ was not eventually loaded. If at the start of document, ⟨package⟩ has not been loaded, a warning message is entered in the log file. Use the following \@ensurepackageloaded macro if you really need an error message in this case.

Macro

```
782 \@ensurepackageloaded{⟨packages⟩}
```

This will issue an error at start of document if any member of the comma-separated list ⟨packages⟩ wasn't loaded before then. This command can be used to signpost those packages that must be loaded later.

Commands restricted to package and preamble

Macro

```
783 \SKV@onlypreamble{⟨list⟩}
784 \SKV@onlypackage
```

The LaTeX kernel's macro \@onlypreamble accepts only one command at a time (i.e., you can't give it a list of preamble commands in one go), and the error message \@notprerr is not that precise, since it doesn't indicate the command that has been wrongly placed in the document's body. The use of \@onlypreamble in a style or class file can be monotonous if the file has many preamble commands. The macro \SKV@onlypreamble takes a no-comma ⟨list⟩ of commands at once

and gives precise error messages related to the incorrectly located commands. The ⟨list⟩ may be populated with nil, one, or more control sequences, e.g.,

Example

```
\SKV@onlypreamble{\macroa \macrob \macroc}
```

All preamble commands can be collected together in one \SKV@onlypreamble, preferably at the end of the style or class file.

The function \SKV@onlypackage may be used to restrict commands to packages only. For example, the following restricts the command \x to packages only:

Example

```
\def\x#1{\SKV@onlypackage\usearg{#1}}
```

Extended \aftergroup and \afterassignment

Macro

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

TEX's \aftergroup and \afterassignment don't accept arbitrary code. The commands \SKV@aftergroup and \SKV@afterassignment execute the arbitrary ⟨code⟩ after a group or assignment. The starred variants expand ⟨code⟩ once before the assignment or before exiting the group. These commands don't accumulate the group and assignment counters indefinitely: the counters are initialized after each group or each assignment.

Some examples follow:

Example

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

\def\aa{aaa} \def\bb{bbb} \def\xx{xxx} \def\yy{yyy}

\begin{document}
\begingroup
  \SKV@aftergroup{\par\aa***\bb}%
  \SKV@aftergroup{\par\bb***\aa}%
```

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

List processing

Macro

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

These are fast for-loops that accept general list parsers and allow for list breaks, as well as give the remainder of the list if a break occurs within the list. Elements of ⟨list⟩ are stored in ⟨cmd⟩, and ⟨function⟩ is executed for each element of ⟨list⟩. ⟨list⟩, which is populated by parser-separated elements, is not expanded before the iteration; ⟨listcmd⟩, on the other hand, is expanded once before the commencement of the loop. The list parser is dynamically declarable via

Macro

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

Also, these iteration macros use the more powerful \SKV@ifblank to check whether or not ⟨list⟩ is empty or blank. The commands \SKV@for@a and \SKV@for@b are robust, but in expansion contexts, both ⟨cmd⟩ and ⟨function⟩ will need to be protected. The ⟨parser⟩ persists in effect until it is changed by another call to \SKV@CommandGenParser or \skvoptions as above.

Note: One snag with a generic list parser like \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 results, since a previous call to \SKV@CommandGenParser might have set a parser that is no longer valid. To obviate this type of situation, the following commands are also provided in the skeyval package:

Macro

```
\SKV@for[⟨parser⟩]{⟨list⟩}⟨cmd⟩{⟨function⟩}
\SKV@for*[⟨parser⟩]⟨listcmd⟩⟨cmd⟩{⟨function⟩}
```

The ⟨parser⟩ appears as an optional argument in these commands and its default value is "," (comma). These commands allow the user to provide the ⟨parser⟩ with every call. The unstarred and starred versions of \SKV@for are equivalent to \SKV@for@a and \SKV@for@b, respectively. Both sets (\SKV@for@a and \SKV@for@b | \SKV@for and \SKV@for*) may be needed in different circumstances. In applications where the ⟨parser⟩ is fixed, the commands \SKV@for@a

and \SKV@for@b are faster than \SKV@for and \SKV@for⋆ because in the former cases the ⟨parser⟩ would then need to be set only once: each call to \SKV@for or \SKV@for⋆, whether or not the optional ⟨parser⟩ is provided, resets the ⟨parser⟩.

The list parser itself is available in \parser, which can be used in ⟨function⟩. An example follows:

Example

```
\SKV@CommandGenParser{;}
\SKV@for@a{a;b;c;d}\cmd{\if a\cmd ‘\cmd’\ is ‘a’\parser
\else\space ‘\cmd’\ isn’t ‘a’\parser\fi}.
```

This list can be broken after, say, elements “a” and “b”, as follows:

Example

```
\SKV@for@a{a;b;listbreak;c;d}\cmd
{\if a\cmd ‘\cmd’\ is ‘a’\parser\else\space ‘\cmd’\
isn’t ‘a’\parser\fi},
```

upon which the remainder of the list is accessible from \SKV@remainder.

Macro

```
\SKV@tfor@a{⟨list⟩}⟨cmd⟩{⟨function⟩}
\SKV@tfor@b⟨listcmd⟩⟨cmd⟩{⟨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 these two macros have been prompted by the following rationale. 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.

The \@break@tfor of the LaTeX kernel allows the user to break out of the \@tfor loop but provides no mechanism for saving the remainder of the ⟨list⟩ upon breaking the list. Secondly, I have had trouble breaking out of simple \@tfor loops. For example, the following fails: LaTeX complains of “extra “fi”, the reason being obvious.

Example

```
\def\one{One}\def\two{Two}\def\three{Three}
\@tfor\x:=\one\two\@break@tfor\three\do{\x}
```

Thirdly, if the content of \x above is sanitized/detokenized in the loop before being used in ⟨function⟩ (or sanitized in the ⟨function⟩ itself), then \@break@tfor can’t break the loop. Consider the following:

Example

```
\@tfor\x:=\one\two\@break@tfor\three\do{%
  \edef\x{\detokenize\expandafter{\x}}%
}
```

Clearly, \@break@tfor can't break this loop. The macros \SKV@tfor@a and \SKV@tfor@b circumvent these problems. Additionally, they (a) reorder the arguments such that ⟨list⟩ comes before ⟨cmd⟩, and (b) remove the need for the usual delimitating tokens, thereby making their syntaxes mimic those of \SKV@for@a and \SKV@for@b.

In the following, the remainder of the list (namely, \three) can be accessed from the macro \SKV@remainder:

Example

```
\SKV@tfor@a{\one\two\listbreak\three}\x{%
  \edef\x{\detokenize\expandafter{\x}}%
}
```

Macro

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

These are equivalent to \SKV@tfor@a and \SKV@tfor@b respectively.

Hook management

Hooking to user-defined macros

Macro

```
\SKV@appto*+?⟨cs⟩{⟨content⟩}
```

This appends ⟨content⟩ to ⟨cs⟩. If ⟨cs⟩ was previously undefined, it is initialized with ⟨content⟩. The star (*) sign directs \SKV@appto to expand ⟨content⟩ once before appending ⟨content⟩ to ⟨cs⟩. The plus (+) sign means that ⟨cs⟩ is a control sequence name instead of a control sequence, while the question mark (?) instructs \SKV@appto to append ⟨content⟩ to ⟨cs⟩ globally (to escape local groups). This command is robust, but fragile arguments must be protected in expansion contexts.

Except for the initialization of undefined ⟨cs⟩, \SKV@appto? is equivalent to LaTeX's \g@addto@macro.

The following non-generic, less powerful, forms of \SKV@appto are also available, but they don't have the starred (*) variants:

Macro

```
\apptomac⟨cs⟩{⟨content⟩}
        → \SKV@appto⟨cs⟩{⟨content⟩}
\gapptomac⟨cs⟩{⟨content⟩}
        → \SKV@appto?⟨cs⟩{⟨content⟩}
\csapptomac{⟨name⟩}{⟨content⟩}
        → \SKV@appto+{⟨name⟩}{⟨content⟩}
\csgapptomac{⟨name⟩}{⟨content⟩}
        → \SKV@appto+?{⟨name⟩}{⟨content⟩}
```

Macro

```
\SKV@prepto*+?⟨cs⟩{⟨content⟩}
```

This prepends ⟨content⟩ to ⟨cs⟩. If ⟨cs⟩ was previously undefined, it is initialized with ⟨content⟩. The star (*) sign directs \SKV@prepto to expand ⟨content⟩ once before prepending it to ⟨cs⟩. The plus (+) sign means that ⟨cs⟩ is a control sequence name instead of a control sequence, while the question mark (?) instructs \SKV@prepto to prepend ⟨content⟩ to ⟨cs⟩ globally (to escape local groups). This command is robust, but fragile arguments must be protected in expansion contexts.

Again, the following non-generic versions of \SKV@prepto are available, but they don't have the starred (*) variants:

Macro

```
\preptomac⟨cs⟩{⟨content⟩}
        → \SKV@prepto⟨cs⟩{⟨content⟩}
\gpreptomac⟨cs⟩{⟨content⟩}
        → \SKV@prepto?⟨cs⟩{⟨content⟩}
\cspreptomac{⟨name⟩}{⟨content⟩}
        →  \SKV@prepto+{⟨name⟩}{⟨content⟩}
\csgpreptomac{⟨name⟩}{⟨content⟩}
        →  \SKV@prepto+?{⟨name⟩}{⟨content⟩}
```

Macro

```
\SKV@addtolist+?<[⟨parser⟩]⟨csa⟩⟨csb⟩
```

This adds the contents of the macro ⟨csb⟩ to the list in the container ⟨csa⟩. The plus sign (+) means that ⟨csa⟩ is a control sequence name (instead of a control sequence); the question mark (?) directs \SKV@addtolist to add the contents of ⟨csb⟩ to ⟨csa⟩ globally (to escape local groups); and the left angle (<), if present, instructs \SKV@addtolist to prepend the contents of ⟨csb⟩ to the left of the contents of ⟨csa⟩, instead of appending to the right by default. The optional argument ⟨parser⟩ is the list parser, ie, the separator of the instances in ⟨csa⟩. Its default value is "," (comma).

Package and document hooks

Macro

```
\SKV@AtPackageEnd{⟨code⟩}
\SKV@AtClassEnd{⟨code⟩}
```

These are the robust versions of the well-known LaTeX hooks \AtEndOfPackage and \AtEndOfClass.

Macro

```
\SKV@BeforePackageEnd{⟨code⟩}
\SKV@BeforeClassEnd{⟨code⟩}
```

```
\SKV@AfterPackageEnd{⟨code⟩}
\SKV@AfterClassEnd{⟨code⟩}
```

The first two of these hook to just before \AtEndOfPackage or \AtEndOfClass; the third and fourth hook to just after \AtEndOfPackage or \AtEndOfClass. They are all robust. These commands have been necessitated by some tasks in the skeyval package, but may be useful in some other contexts.

Macro

```
\SKV@AtDocumentStart{⟨code⟩}
```

This is the robust version of the LaTeX hook \AtBeginDocument. It can be used in expansion contexts without protection, but fragile arguments within it must be protected.

Macro

```
\SKV@BeforeDocumentStart{⟨code⟩}
\SKV@AfterDocumentStart{⟨code⟩}
```

These provide two more document hooks. They are both robust. The command \SKV@BeforeDocumentStart differs from \AtBeginDocument in that the former is executed right at the end of the preamble, before the main auxiliary file (as written on the previous LaTeX run) is read and prior to the execution of any \AtBeginDocument code. It isn't possible to write to the auxiliary file at the point \SKV@BeforeDocumentStart is executed.

\SKV@AfterDocumentStart differs from \AtBeginDocument in the sense that the former is executed at the tag end of \begin \{document\}, after the execution of any \AtBeginDocument code. Commands whose scope are restricted to the document's preamble with \@onlypreamble or \SKV@onlypreamble are no longer committable when \SKV@AfterDocumentStart is being executed.

Macro

```
\SKV@AtDocumentEnd{⟨code⟩}
\SKV@BeforeLastPage{⟨code⟩}
\SKV@AfterLastPage{⟨code⟩}
\SKV@AfterDocumentEnd{⟨code⟩}
```

The last three of these are wrappers developed based on the atveryend package, which provides a consistent mechanism for \enddocument methods. The macro \SKV@AtDocumentEnd is the robust equivalent of \AtEndDocument. The macro \SKV@BeforeLastPage appends ⟨code⟩ after the \@enddocumenthook but before the last \clearpage, and thus before the last shipout. The command \SKV@AfterLastPage executes ⟨code⟩ after the last \clearpage invoked within \enddocument, i.e., after the last shipout but before the main auxiliary file is closed. This is, e.g., the right instance to record the last document page in the auxiliary file.

The command \SKV@AfterDocumentEnd appends ⟨code⟩ to the very end of the document, after all of the end-of-document codes have been executed (i.e., after

the main .aux file of the current pass has been read and all \AtEndDocument codes have been effected, except font and label/reference warnings).

Inputting files

Macro

```
\InputFileOnce[⟨path⟩]{⟨file⟩}
\InputFileOnce*[⟨path⟩]{⟨file⟩}
```

The unstarred variant of this command inputs ⟨file⟩ on ⟨path⟩ but only once in one LaTeX pass. The argument ⟨path⟩ is optional and its default value is the current/document's environment/directory. If ⟨file⟩ had previously been read, a warning message is entered in the transcript log file (if the package option verbose is selected) and the input is aborted. If ⟨file⟩ doesn't exist on ⟨path⟩, as many as desired opportunities are given to the user to type in the correct filename on the screen, or enter "no" to continue with the LaTeX pass without inputting the file.

If the user wants to input ⟨file⟩ more than once in one document, then he should use the starred (⋆) variant of \InputFileOnce.

15 References

All the (LA)TeX packages cited in this guide are available on CTAN (the Comprehensive TeX Archive Network).

16 Version history

The following change history highlights significant changes that affect user utilities and interfaces; mutations of technical nature are not documented in this section. The numbers on the right-hand side of the following lists are section numbers; the star sign (⋆) means the subject features in the package but is not reflected anywhere in this user guide.

Version 0.1 [2010/01/01]

First public release under the name keyreader package . . . . . ⋆

Introduced complementary native-boolean keys 4

Provided machinery for reading multiple keys of all kinds from just one command 7

Version 0.2 [2010/01/10]

Fixed a bug and optimized the \define@keylist loop 7

Version 0.3 [2010/01/20]

Introduced toggles and toggle-boolean keys 5

Version 0.4 [2010/02/01]

Introduced complementary toggle-boolean keys 6

Version 0.5 [2010/02/23]

Changed the name of the package from keyreader package to skeyval package . . . . . ⋆

Introduced the following macros 9.1

```
\ifkeydefined, \ifkeyreserved, \ifkeysuspended
```

Provided the following facilities 9.2.1

```
\newordkey, \newcmdkey, \newboolkey, \newchoicekey,
\newtogkey, \define@biboolkeys, \newbiboolkeys,
\define@bitogkeys, \newbitogkeys, \define@uniboolkeys,
\newuniboolkeys, \define@unitogkeys, \newunitogkeys,
\define@biuniboolkeys, \newbiuniboolkeys,
\define@biunitogkeys, \newbiunitogkeys
```

Provided mechanisms for disabling, localizing, reserving, unreserving, suspending, restoring, and removing keys 10

Redefined a few of xkeyval package's internal macros . . . . . ⋆

Normalization of key-value lists before parsing 13

Included some developer macros 14

Version 0.6 [2010/03/30]

Modified the mechanics of the internal macros of the following 10.3

```
\ReserveKeyPrefixNames, \ReserveKeyPrefixNames⋆,
\ReserveMacroPrefixNames, \ReserveMacroPrefixNames⋆,
\ReserveFamilyNames, \ReserveFamilyNames⋆
```

Introduced the following macros 14

```
\SKV@AtPackageEnd, \SKV@BeforePackageEnd,
\SKV@AfterPackageEnd, \SKV@AtDocumentEnd,
\SKV@BeforeLastPage, \SKV@AfterLastPage,
\SKV@AfterDocumentEnd, \SKV@onlypreamble,
\SKV@onlypackage, \SKV@ifdraft, \SKV@iffinal,
\SKV@ifpdf, \@ensurepackageloaded, \InputFileOnce,
\newswitch, \defswitch, etc.
```

Alert the user if skeyval package is loaded before \documentclass . . 3

Introduced "user-value" keys 11

Extended the pointer system of the xkeyval package 12

Corrected a bug in the xkeyval package which, if \setkeys is nested in a class file, gives non-empty \@unusedoptionlist even if all the options of \documentclass have been used . . . . . ⋆

Version 0.7 [2010/05/05]

Native-boolean and toggle-boolean keys now accept on and off as values, in addition to true and false 5.1

. . . . . 7.4.3

Replaced the former \NewIfs with \NewBooleans and redefined \NewIfs 4.1

Introduced \DeclareKeyCommand . . . . . ⋆

Version 0.71 [2010/05/06]

Corrected a bug in one of the internal conditional tests . . . . . ⋆

Removed \DeclareKeyCommand to create a separate skeycommand package . . . . . ⋆

Index

Index numbers refer to page numbers.

T

U

V

X