The keyreader Package*

Ahmed Musa
a.musa@rocketmail.com

January 26, 2010

Contents

1 Motivation

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

```
1 \newif\ifmyboolean    ->  3 separate commands:
2                           \if<myboolean> <myboolean>true
```

*Version 0.3.

```
                          <myboolean>false

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

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

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

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

2 Package loading

The package can be loaded in style and class files by

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

and in document files via

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

where the options and their default values are

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

The parser is the separator between the keys in the key list to be defined in one go (see examples in section 5.3). All these options can be set dynamically by using the \krsetup macro:

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

3 Complementary boolean keys

The syntax of complementary boolean keys is

```
\define@comp@boolkeys[<key-prefix>]{<family>}[<macro-prefix]
  {<primary boolean>}[<default value for primary boolean>]
  {<secondary boolean>}{<func for primary boolean>}
  {<func for secondary boolean>}.
```

When the user doesn't supply the <key-prefix> and/or <macro-prefix>, the package will use <KV> and <mp@>, respectively. When one boolean (primary or secondary) is true, the other is automatically set false. Infinite loops, which are possible in back-linked key settings, are avoided in the keyreader package.

As an example, we define below two complementary keys <draft> and <final> with different key macros:

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

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

```
\define@comp@boolkeys{fam}{draft}[true]{final}{}{}.
```

The defined complementary keys <draft> and <final> can now be set as follows:

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

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

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

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

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

4 Toggle switches and keys

4.1 Toggle switches

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

```
28 \deftog{<toggle>}
```

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

```
29 \newtog{<toggle>}
```

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

```
30 \providetog{<toggle>}
```

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

```
31 \requiretog{<toggle>}
```

\requiretog takes arguments like \newtog and behaves like \providetog with the difference: if the toggle is already defined, the command \requiretog calls

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

The keyreader 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.

```
32 \settog{<toggle>}{<true | false>}
```

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

```
33 \togtrue{<toggle>}
```

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

```
34 \togfalse{<toggle>}
```

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

```
35 \iftog{<toggle>}{<true>}{<false>}
```

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

```
36 \ifnottog{<toggle>}{<not true>}{<not false>}
```

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

4.2 Toggle keys

The syntax for defining toggle keys is exactly like those for boolean keys in the xkeyval package:

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

If the macro prefix <mp> is not specified, these create a toggle of the form <pre>@<family>@<key> using \newtog (which initializes the switch to <false>) and a key macro of the form \<pre>@<family>@<key> which first checks the validity of the user input. If the value is valid, it uses it to set the boolean and then executes <func>. If the user input wasn't valid, then the boolean will not be set and the package will generate a fatal error.

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 \<pre>@<family>@<key>. The value <default> will be used by the key macro when the user sets the key without a value.

If the + version of the macro is used, the user can specify two key macros <func1> and <func2>.

If user input is valid, the macro will set the toggle and executes <func1>; otherwise, it will not set the boolean but will execute <func2>.

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

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

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

```
}
```

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

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

```
\define@togkeys[<pre>]{<fam>}[<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 5 for how to define multiple keys with custom functions.

As an example of defining multiple toggle keys, consider

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

This is an abbreviation for

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

Now we can do

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

\krsetkeys[KV]{fam}{book=true}
```

Toggle keys can be set in the same way that other key types are set. \krsetkeys is introduced by the keyreader package as a drop-in replacement for the legacy \setkeys of the xkeyval package (see Section 7.2). \setkeys can still be used when the situation permits (see Section 7.2).

The status of toggles can be examined by doing

```
\show\<KR@toggle@><mp><key>
```

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

```
\show\<KR@toggle@><pre>@<family>@<key>.
```

5 Defining multiple keys by one command

The main user interface for defining multiple keys is the \define@keylist macro, whose syntax is

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

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

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

```
\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:

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

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

5.1 Choice key values

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

```
\ChoiceKeyValues{<key>}{<comma-separated list of admissible
   key values>}.
```

To further save tokens, the macro \ChoiceKeyValues may be abbreviated by \CKVS. It has to be defined each time a choice key is being defined. For example,

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

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

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

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

```
\krsetup{parser=value,macroprefix=value,keyprefix=value,
  keyfamily=value,xchoicelist=value}.
```

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

5.2 Internals

The internal equivalent of `\ChoiceKeyValues` (the choice key list of alternative values) is the macro `\<family@key@altlist>`. For example, for the `align` key above, the internal of `\CKVS` is `\fam@align@altlist`, assuming the current family is `fam`.

For all keys in a family, the internal of the key macro/function is available in `\<family@key@func>`, and the value submitted by the user when setting the key can be accessed via the macro `\<family@key@value>`.

5.3 Examples

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

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

Further, suppose we wish to define a set of keys `<color,angle,scale,align>`. The keys `color`, `angle` and `scale` will de defined using command keys, while the key `align` will be defined by choice keys. Assume that the `align` key can only assume 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`, where `\do` is assumed defined elsewhere. The keys `color` and `angle` aren't associated with macros. Then we can go:

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

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

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

Instead of defining the macro `\f@align` before hand, we can submit its replacement text directly to the macro `\define@keylist`, but, because `\f@align` contains a conditional, some care is needed in doing so (see section 7). Once the key `align` has been defined, the macro `\f@align` can be reused—perhaps to define other keys—even before the key `align` is set. This is because it isn't `\f@align` that is used in defining the key `align` but its internal counterpart

(i.e., a family-dependent internal of \f@align, which is \fam@align@func). In this way, the user can economize on tokens. The same applies to all the macros that may be used in defining keys.

Note the \stopread command inserted above. Because of it, the key mybool will not be read and defined; the rest (i.e., color, angle, scale and align) will be read and defined. All the entries for mybool and weather will instead be saved in the macro \KR@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 page setup keys:

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

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

which have the following trivial key macros:

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

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

\def\f@paperwidth{\wlog{‘paperwidth’ was defined as %
  ordinary key.}}
```

```
\newcommand\f@align{%
  \ifcase\nr\relax
    \def\mp@align{\centering}%
  \or
    \def\mp@align{\flushright}%
  \or
    \def\mp@align{\flushleft}%
  \or
    \let\mp@align\relax
  \fi
}
```

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

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

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

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

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

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

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

holds the macros:

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

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

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

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

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

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

```
\toks0={\long\def\f@testdim#1{A test dimension #1\par\bigskip}}

\define@keylist{3,boolvar,true,;2,testdim,2cm,\the\toks0}.
```

The advantage of using token registers is that the parameter characters need not be doubled in the token registers, unlike when using macros. The token register \toks 0 can be reused as soon as the key testdim is defined.

5.4 Demonstrating the effect of limiting \textwidth

After defining and setting the keys above, the following tokens can be used in a source file to demonstrate the effect of setting the page setup keys shown above:

```
\begin{center}
  \begin{minipage}{\mp@textwidth}
    \lipsum[1]
      % From the lipsum package.
      % The blindtext package can also be used.
  \end{minipage}
\end{center}

\lipsum[1]
```

6 Input error

Both boolean and choice keys issue error messages if the input is not valid, i.e., not in the list of admissible values. The default input error is defined by `\KR@inputerr` macro to be

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

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

7 Conditionals in key macros

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

7.1 Burying conditionals in macros or token registers

Key macros/functions involving conditional operations such as

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

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

Suppose we want to submit the following:

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

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

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

and then do

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

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

As another example, we may do

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

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

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

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

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

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

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

You can see the significant reduction in the number of parameter characters when using token registers. The token registers `\toks 0` and `\toks 1` can be reused to define many other keys.

7.2 Using a "dirty" trick to submit the conditionals

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

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

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

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

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

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

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

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

Now, however, when setting the key `mybool`, the user has to use `\krsetkeys` instead of `xkeyval`'s legacy `\setkeys`. The command `\krsetkeys` does understand that `\fif` and `\ffi` stand for `\if` and `\fi`, respectively, and have been used to "deceive" TEX. `\krsetkeys` has the same syntax as `\setkeys`:

```
\krsetkeys*+[key prefix]{key family}{keys=values}.
```

`\krsetkeys` can in general be used in place of `\setkeys`, even in instances (i.e., for keys) where `\fif` and `\ffi` have not been used.

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

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

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

7.3 Using toggles

Toggle switches, described in Section 4, can also be used to circumvent the problem of matching `\if` and `\fi` in difficult circumstances, since toggles aren't

TEX primitives. For example, the following works:

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

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

8 Disabling keys

The keyreader package has modified the definition of \disable@keys from the xkeyval package to allow for bespoke warnings and error messages, without engendering any conflict with the legacy \disable@keys. The new command is \krdisable@keys; the use syntax remains the same as that of \disable@keys:

```
\krdisable@keys[<key prefix>]{<key family>}{<comma %
  separated list of keys to disable>}.
```

Any attempt to subsequently set 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" \KR@disabledkey@err and \KR@disabledkey should be retained.

```
\def\KR@disabledkey@err{%
  \PackageError{keyreader}{%
    Key `\KR@disabledkey' has been disabled.}{%
    You can't set or reset it at this late stage.\MessageBreak
    You should have set it earlier in the\MessageBreak
    \string\documentclass\space or \string\usepackage
  }%
}
```

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

9 Epilogue

The are many commands available in the package for general use, but they are not documented here.