The elatex package

An e-TeX toolbox for class and package authors

Philipp Lehman
plehman@gmx.net

Version 1.2
July 13, 2007

Contents

1 Introduction

1.1 About

The elatex package is a toolbox of programming facilities geared primarily towards LaTeX class and package authors. It provides LaTeX frontends to some of the new primitives provided by e-TeX as well as some generic tools which are not strictly related to e-TeX but match the profile of this package. This package will not modify any part of the LaTeX kernel. Its name is not meant to imply that it patches LaTeX such that the kernel makes use of e-TeX facilities by default. The package is work in progress.

1.2 License

1.3 Contributions and feedback

I started to work on this package when I found myself implementing the same tools and shorthands I had employed in previous LaTeX packages for yet another package. For the most part, the facilities provided by elatex address my needs as a package author and future development is likely to be guided by these needs as well. Please note that I will not be able to address any feature requests. How-

1 http://www.ctan.org/tex-archive/macros/latex/base/lppl.txt

ever, I am open to contributions by other class and package authors, provided that the contributed code is sufficiently generic, matches the profile of this package, and may be added to the package without requiring significant adaption.

1.4 Acknowledgments

The `\ifblank` test of this package is based on code by Donald Arseneau.

2 User commands

The facilities in this section are geared towards regular users as well as class and package authors.

2.1 Definitions

`\newrobustcmd{`⟨*command*⟩`}[`⟨*arguments*⟩`][`⟨*optarg default*⟩`]{`⟨*definition*⟩`}`

The syntax and behavior of this command is similar to `\newcommand` except that the newly defined ⟨*command*⟩ is robust. This command differs from the `\DeclareRobustCommand` command from the LaTeX kernel in that it issues an error rather than just an informational message if the ⟨*command*⟩ is already defined. Since it uses e-TeX's low-level protection mechanism rather than the corresponding high-level LaTeX facilities, it does not require an additional macro to implement the 'robustness'. This command itself is also robust.

`\renewrobustcmd{`⟨*command*⟩`}[`⟨*arguments*⟩`][`⟨*optarg default*⟩`]{`⟨*definition*⟩`}`

The syntax and behavior of this command is similar to `\renewcommand` except that the redefined ⟨*command*⟩ is robust. This command itself is also robust.

`\providerobustcmd{`⟨*command*⟩`}[`⟨*arguments*⟩`][`⟨*optarg default*⟩`]{`⟨*definition*⟩`}`

The syntax and behavior of this command is similar to `\providecommand` except that the newly defined ⟨*command*⟩ is robust. Note that this command only provides a robust definition if the ⟨*command*⟩ is undefined. It will not make an already defined ⟨*command*⟩ robust. This command itself is robust.

2.2 Patching

`\robustify{`⟨*command*⟩`}`

Redefines a ⟨*command*⟩ such that it is robust without altering its syntax or definition. If the ⟨*command*⟩ has been defined with `\DeclareRobustCommand`, this will be detected automatically. LaTeX's high-level protection mechanism is replaced by the corresponding low-level e-TeX facility in this case. This command is robust and may only be used in the document preamble.

3 Author commands

The facilities in this section are geared towards class and package authors.

3.1 Definitions

The facilities in this section are simple but frequently required shorthands which extend the scope of the `\@namedef` and `\@nameuse` macros from the LaTeX kernel.

`\csdef{`⟨*csname*⟩`}`⟨*arguments*⟩`{`⟨*definition*⟩`}`

Similar to the TeX primitive `\def` except that it takes a control sequence name as its first argument. This command is robust and corresponds to `\@namedef`.

`\csgdef{`⟨*csname*⟩`}`⟨*arguments*⟩`{`⟨*definition*⟩`}`

Similar to the TeX primitive `\gdef` except that it takes a control sequence name as its first argument. This command is robust.

`\csedef{`⟨*csname*⟩`}`⟨*arguments*⟩`{`⟨*definition*⟩`}`

Similar to the TeX primitive `\edef` except that it takes a control sequence name as its first argument. This command is robust.

`\csxdef{`⟨*csname*⟩`}`⟨*arguments*⟩`{`⟨*definition*⟩`}`

Similar to the TeX primitive `\xdef` except that it takes a control sequence name as its first argument. This command is robust.

`\protected@csedef{`⟨*csname*⟩`}`⟨*arguments*⟩`{`⟨*definition*⟩`}`

Similar to `\csedef` except that LaTeX's protection mechanism is temporarily enabled. To put it in other words: this command is similar to the LaTeX kernel command `\protected@edef` except that it takes a control sequence name as its first argument. This command is robust.

`\protected@csxdef{`⟨*csname*⟩`}`⟨*arguments*⟩`{`⟨*definition*⟩`}`

Similar to `\csxdef` except that LaTeX's protection mechanism is temporarily enabled. To put it in other words: this command is similar to the LaTeX kernel command `\protected@xdef` except that it takes a control sequence name as its first argument. This command is robust.

`\cslet{`⟨*csname*⟩`}{`⟨*command*⟩`}`

Similar to the TeX primitive `\let` except that the first argument is a control sequence name. This command is robust.

`\letcs{`⟨*command*⟩`}{`⟨*csname*⟩`}`

Similar to the TeX primitive `\let` except that the second argument is a control sequence name. This command is robust.

`\csletcs{`⟨*csname*⟩`}{`⟨*csname*⟩`}`

Similar to the TeX primitive `\let` except that both arguments are control sequence names. This command is robust.

`\csuse{`⟨*csname*⟩`}`

Takes a control sequence name as its argument and forms a control sequence token. This command differs from the `\@nameuse` macro from the LaTeX kernel in that it expands to an empty string if the control sequence is undefined.

3.2 Arithmetic definitions

The facilities in this section permit calculations using macros rather than length registers and counters.

`\numdef{`⟨*command*⟩`}{`⟨*integer expression*⟩`}`

This command is similar to `\edef` except that the ⟨*integer expression*⟩ is processed with `\numexpr`. The ⟨*integer expression*⟩ may be any arbitrary code which is valid in this context. The definition assigned to ⟨*command*⟩ will be the result of that calculation. If the ⟨*command*⟩ is undefined, it will be initialized to '0' before the ⟨*integer expression*⟩ is processed.

`\numgdef{`⟨*command*⟩`}{`⟨*integer expression*⟩`}`

Similar to `\numdef` except that the assignment is global.

`\csnumdef{`⟨*csname*⟩`}{`⟨*integer expression*⟩`}`

Similar to `\numdef` except that it takes a control sequence name as its first argument.

`\csnumgdef{`⟨*csname*⟩`}{`⟨*integer expression*⟩`}`

Similar to `\numgdef` except that it takes a control sequence name as its first argument.

`\dimdef{`⟨*command*⟩`}{`⟨*dimen expression*⟩`}`

This command is similar to `\edef` except that the ⟨*dimen expression*⟩ is processed with `\dimexpr`. The ⟨*dimen expression*⟩ may be any arbitrary code which is valid in this context. The definition assigned to ⟨*command*⟩ will be the result of that calculation. If the ⟨*command*⟩ is undefined, it will be initialized to '`0pt`' before the ⟨*dimen expression*⟩ is processed.

`\dimgdef{`⟨*command*⟩`}{`⟨*dimen expression*⟩`}`

Similar to `\dimdef` except that the assignment is global.

`\csdimdef{`⟨*csname*⟩`}{`⟨*dimen expression*⟩`}`

Similar to `\dimdef` except that it takes a control sequence name as its first argument.

`\csdimgdef{`⟨*csname*⟩`}{`⟨*dimen expression*⟩`}`

Similar to `\dimgdef` except that it takes a control sequence name as its first argument.

`\gluedef{`⟨*command*⟩`}{`⟨*glue expression*⟩`}`

This command is similar to `\edef` except that the ⟨*glue expression*⟩ is processed with `\glueexpr`. The ⟨*glue expression*⟩ may be any arbitrary code which is valid in this context. The definition assigned to ⟨*command*⟩ will be the result of that calculation. If the ⟨*command*⟩ is undefined, it will be initialized to '`0pt plus 0pt minus 0pt`' before the ⟨*glue expression*⟩ is processed.

`\gluegdef{`⟨*command*⟩`}{`⟨*glue expression*⟩`}`

Similar to `\gluedef` except that the assignment is global.

`\csgluedef{`⟨*csname*⟩`}{`⟨*glue expression*⟩`}`

Similar to `\gluedef` except that it takes a control sequence name as its first argument.

`\csgluegdef{`⟨*csname*⟩`}{`⟨*glue expression*⟩`}`

Similar to `\gluegdef` except that it takes a control sequence name as its first argument.

`\mudef{`⟨*command*⟩`}{`⟨*muglue expression*⟩`}`

This command is similar to `\edef` except that the ⟨*muglue expression*⟩ is processed with `\muexpr`. The ⟨*muglue expression*⟩ may be any arbitrary code which is valid in this context. The definition assigned to ⟨*command*⟩ will be the result of that calculation. If the ⟨*command*⟩ is undefined, it will be initialized to '`0mu`' before the ⟨*muglue expression*⟩ is processed.

`\mugdef{`⟨*command*⟩`}{`⟨*muglue expression*⟩`}`

Similar to `\mudef` except that the assignment is global.

`\csmudef{`⟨*csname*⟩`}{`⟨*muglue expression*⟩`}`

Similar to `\mudef` except that it takes a control sequence name as its first argument.

`\csmugdef{`⟨*csname*⟩`}{`⟨*muglue expression*⟩`}`

Similar to `\mugdef` except that it takes a control sequence name as its first argument.

3.3 Expansion control

The facilities in this section are useful to control expansion in an `\edef` or a similar context.

`\expandonce{`⟨*command*⟩`}`

This command expands ⟨*command*⟩ once and prevents further expansion of the replacement text.

`\csexpandonce{`⟨*csname*⟩`}`

Similar to `\expandonce` except that it takes a control sequence name as its argument.

`\protecting{`⟨*code*⟩`}`

This command applies LaTeX's protection mechanism, which normally requires prefixing each fragile command with `\protect`, to an entire chunk of arbitrary ⟨*code*⟩ or text. Its behavior depends on the current state of `\protect`. Note that the braces around ⟨*code*⟩ are mandatory even if it is a single token.

3.4 Hook management

The facilities in this section are intended for hook management. A 'hook' in this context is a plain macro without any arguments and prefixes which is used to collect code to be executed later. These facilities may also be useful to patch simple macros by appending code to them. For more complex patching operations, see section 3.5. All commands in this section will initialize the hook if it is undefined.

3.4.1 Appending code to a hook

The facilities in this section append arbitrary code to a hook.

`\appto{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command appends arbitrary ⟨*code*⟩ to a ⟨*command*⟩. If the ⟨*code*⟩ contains any parameter characters, they need not be doubled. This command is robust.

`\gappto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\appto` except that the assignment is global. This command may be used as a drop-in replacement for the `\g@addto@macro` command in the LaTeX kernel.

`\eappto{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command appends arbitrary ⟨*code*⟩ to a ⟨*command*⟩. The ⟨*code*⟩ is expanded at definition-time. Only the new ⟨*code*⟩ is expanded, the current definition of ⟨*command*⟩ is not. This command is robust.

`\xappto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\eappto` except that the assignment is global.

`\protected@eappto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\eappto` except that LaTeX's protection mechanism is temporarily enabled.

`\protected@xappto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\xappto` except that LaTeX's protection mechanism is temporarily enabled.

`\csappto{`⟨*csname*⟩`}{`⟨*code*⟩`}`

Similar to `\appto` except that it takes a control sequence name as its first argument.

`\csgappto{`⟨*csname*⟩`}{`⟨*code*⟩`}`

Similar to `\gappto` except that it takes a control sequence name as its first argument.

`\cseappto{`⟨*csname*⟩`}{`⟨*code*⟩`}`

Similar to `\eappto` except that it takes a control sequence name as its first argument.

`\csxappto{`⟨*csname*⟩`}{`⟨*code*⟩`}`

Similar to `\xappto` except that it takes a control sequence name as its first argument.

`\protected@cseappto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\protected@eappto` except that it takes a control sequence name as its first argument.

`\protected@csxappto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\protected@xappto` except that it takes a control sequence name as its first argument.

3.4.2 Prepending code to a hook

The facilities in this section 'prepend' arbitrary code to a hook, i. e., the code is inserted at the beginning of the hook rather than being added at the end.

`\preto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\appto` except that the ⟨*code*⟩ is prepended.

`\gpreto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\preto` except that the assignment is global.

`\epreto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\eappto` except that the ⟨*code*⟩ is prepended.

`\xpreto{`⟨*command*⟩`}{`⟨*code*⟩`}`

Similar to `\epreto` except that the assignment is global.

\protected@epreto{⟨*command*⟩}{⟨*code*⟩}

Similar to \epreto except that LaTeX's protection mechanism is temporarily enabled.

\protected@xpreto{⟨*command*⟩}{⟨*code*⟩}

Similar to \xpreto except that LaTeX's protection mechanism is temporarily enabled.

\cspreto{⟨*csname*⟩}{⟨*code*⟩}

Similar to \preto except that it takes a control sequence name as its first argument.

\csgpreto{⟨*csname*⟩}{⟨*code*⟩}

Similar to \gpreto except that it takes a control sequence name as its first argument.

\csepreto{⟨*csname*⟩}{⟨*code*⟩}

Similar to \epreto except that it takes a control sequence name as its first argument.

\csxpreto{⟨*csname*⟩}{⟨*code*⟩}

Similar to \xpreto except that it takes a control sequence name as its first argument.

\protected@csepreto{⟨*command*⟩}{⟨*code*⟩}

Similar to \protected@epreto except that it takes a control sequence name as its first argument.

\protected@csxpreto{⟨*command*⟩}{⟨*code*⟩}

Similar to \protected@xpreto except that it takes a control sequence name as its first argument.

3.5 Patching

The facilities in this section are useful to hook into or modify existing code. All commands presented here preserve the number of arguments and the prefixes of the patched ⟨*command*⟩. Note that the patching process involves detokenizing the ⟨*command*⟩ and retokenizing it under the current category code regime after patching. The category code of '@' is temporarily set to 11. If the definition of the ⟨*command*⟩ includes any tokens with non-standard category codes, the respective category codes must be adjusted prior to patching. If the code to be replaced or inserted refers to the parameters of the ⟨*command*⟩ to be patched, the parameter characters need not be doubled when invoking one of the commands below. Note that \outer commands may not be patched.

`\patchcmd`[⟨*prefix*⟩]{⟨*command*⟩}{⟨*search*⟩}{⟨*replace*⟩}{⟨*success*⟩}{⟨*failure*⟩}

This command extracts the definition of a ⟨*command*⟩, replaces ⟨*search*⟩ with ⟨*replace*⟩, and reassembles the ⟨*command*⟩. The pattern match is category code agnostic and matches the first occurence of the ⟨*search*⟩ string in the definition of the ⟨*command*⟩ to be patched. If an optional ⟨*prefix*⟩ is specified, it replaces the prefixes of the ⟨*command*⟩. An empty ⟨*prefix*⟩ strips all prefixes from the ⟨*command*⟩. This command executes ⟨*success*⟩ if patching succeeds, and ⟨*failure*⟩ otherwise. It is robust and may only be used in the document preamble. The assignment is local.

`\ifpatchable`{⟨*command*⟩}{⟨*search*⟩}{⟨*true*⟩}{⟨*false*⟩}

This command executes ⟨*true*⟩ if the ⟨*command*⟩ is defined and the ⟨*search*⟩ pattern is found in its definition, and ⟨*false*⟩ otherwise. This command is robust and may only be used in the document preamble.

`\apptocmd`{⟨*command*⟩}{⟨*code*⟩}

This command appends ⟨*code*⟩ to the definition of ⟨*command*⟩. In contrast to the `\appto` command from section 3.4.1, this one may be used to patch a ⟨*command*⟩ which takes an arbitrary number of arguments. The ⟨*code*⟩ may refer to the parameters of the ⟨*command*⟩ in this case. This command is robust and may only be used in the document preamble. The assignment is local.

`\pretocmd`{⟨*command*⟩}{⟨*code*⟩}

This command is similar to `\apptocmd` except that the ⟨*code*⟩ is 'prepended', i. e., inserted at the beginning of the definition of ⟨*command*⟩. In contrast to the `\preto` command from section 3.4.1, this one may be used to patch a ⟨*command*⟩ which takes an arbitrary number of arguments. The ⟨*code*⟩ may refer to the parameters of the ⟨*command*⟩ in this case. This command is robust and may only be used in the document preamble. The assignment is local.

3.6 Generic tests

`\ifdef`{⟨*command*⟩}{⟨*true*⟩}{⟨*false*⟩}

A LaTeX wrapper for the e-TeX primitive `\ifdefined`. This command expands to ⟨*true*⟩ if the ⟨*command*⟩ is defined, and to ⟨*false*⟩ otherwise. Note that the ⟨*command*⟩ is considered as defined even if its meaning is `\relax`.

`\ifundef`{⟨*command*⟩}{⟨*true*⟩}{⟨*false*⟩}

Expands to ⟨*true*⟩ if the ⟨*command*⟩ is undefined, and to ⟨*false*⟩ otherwise. Apart from reversing the logic of the test, this command also differs from `\ifdef` in that the ⟨*command*⟩ is considered as undefined if its meaning is `\relax`.

`\ifcsdef`{⟨*csname*⟩}{⟨*true*⟩}{⟨*false*⟩}

A LaTeX wrapper for the e-TeX primitive `\ifcsname`. This command expands to ⟨*true*⟩ if ⟨*csname*⟩ is defined, and to ⟨*false*⟩ otherwise. Note that ⟨*csname*⟩ is

considered as defined even if its meaning is `\relax`.

`\ifcsundef{`⟨*csname*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if ⟨*csname*⟩ is undefined, and to ⟨*false*⟩ otherwise. Apart from reversing the logic of the test, this command also differs from `\ifcsdef` in that the ⟨*csname*⟩ is considered as undefined if its meaning is `\relax`. This command may be used as a drop-in replacement for the `\@ifundefined` test in the LaTeX kernel.

`\ifblank{`⟨*string*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the ⟨*string*⟩ is blank (empty or spaces), and to ⟨*false*⟩ otherwise. The ⟨*string*⟩ is not expanded in the test.

3.7 Boolean switches

This package provides two interfaces to boolean switches which are independent of each other. The facilities in section 3.7.1 are a LaTeX frontend to `\newif`. Those in section 3.7.2 use a different mechanism.

3.7.1 Backwards-compatible switches

Since the facilities in this section are based on `\newif` internally, they may be used to test and alter the state of switches previously defined with `\newif`. They are also compatible with the boolean tests of the `ifthen` package. This approach requires a total of three macros per switch.

`\newbool{`⟨*name*⟩`}`

Defines a new boolean switch called ⟨*name*⟩. If the switch has already been defined, this command issues an error.

`\providebool{`⟨*name*⟩`}`

Defines a new boolean switch called ⟨*name*⟩ unless it has already been defined.

`\booltrue{`⟨*name*⟩`}`

Sets the boolean switch ⟨*name*⟩ to `true`.

`\boolfalse{`⟨*name*⟩`}`

Sets the boolean switch ⟨*name*⟩ to `false`.

`\ifbool{`⟨*name*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the state of the boolean switch ⟨*name*⟩ is `true`, and to ⟨*false*⟩ otherwise.

`\notbool{`⟨*name*⟩`}{`⟨*not true*⟩`}{`⟨*not false*⟩`}`

Similar to `\ifbool` but reverses the logic of the test.

3.7.2 Pure LaTeX switches

In contrast to the switches from section 3.7.1, the facilities in this section require only one macro per switch. They also use a separate namespace to avoid name clashes with regular macros.

`\newswitch{`⟨*name*⟩`}`

Defines a new boolean switch called ⟨*name*⟩. If the switch has already been defined, this command issues an error.

`\provideswitch{`⟨*name*⟩`}`

Defines a new boolean switch called ⟨*name*⟩ unless it has already been defined.

`\switchtrue{`⟨*name*⟩`}`

Sets the boolean switch ⟨*name*⟩ to `true`.

`\switchfalse{`⟨*name*⟩`}`

Sets the boolean switch ⟨*name*⟩ to `false`.

`\ifswitch{`⟨*name*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the state of the boolean switch ⟨*name*⟩ is `true`, and to ⟨*false*⟩ otherwise.

`\notswitch{`⟨*name*⟩`}{`⟨*not true*⟩`}{`⟨*not false*⟩`}`

Similar to `\ifswitch` but reverses the logic of the test.

3.8 List processing

`\DeclareListParser{`⟨*command*⟩`}{`⟨*separator*⟩`}`

This command defines a list parser similar to the `\docsvlist` command below, which is defined with '`\DeclareListParser{\docsvlist}{,}`'. Note that the list parsers are sensitive to the category code of the ⟨*separator*⟩.

`\docsvlist{`⟨*item, item, ...*⟩`}`

This command loops over a comma-separated list and executes the auxiliary command `\do` for every item in the list, passing the item as an argument. In contrast to the `\@for` loop in the LaTeX kernel, `\docsvlist` is expandable. With a suitable definition of `\do`, lists may be processed inside an `\edef` or a comparable context. You may use `\listbreak` in the definition of `\do` to stop processing and discard the remaining items in the list. Whitespace after list separators is ignored. If an item contains a comma or starts with a space, it must be wrapped in curly braces. The braces will be removed as the list is processed. Here is a usage example which prints a comma-separated list as an `itemize` environment:

```
\begin{itemize}
  \renewcommand*{\do}[1]{\item #1}
```

```
  \docsvlist{item1, item2, {item3a, item3b}, item4}
\end{itemize}
```

Here is another example:

```
\renewcommand*{\do}[1]{* #1\MessageBreak}
\PackageInfo{mypackage}{%
  Example list:\MessageBreak
  \docsvlist{item1, item2, {item3a, item3b}, item4}}
```

In this example, the list is written to the log file as part of an informational message. The list processing takes place during the `\write` operation.

3.9 Miscellaneous tools

`\rmntonum`{⟨*numeral*⟩}

The primitve TeX command `\romannumeral` converts an integer to a Roman numeral but TeX or LaTeX provide no command which goes the opposite way. The `\rmntonum` command converts a Roman numeral to an integer. The parsing of the numeral is case-insensitive. Any invalid tokens in the ⟨*numeral*⟩ argument are silently ignored and have a value of zero. Since `\rmntonum` is expandable, it may be used in counter assignments, `\ifnum` comparisons, or in an `\edef`. For example:

```
\setcounter{counter}{\rmntonum{CXVI}}
\ifnum\rmntonum{mcmxcviii}>1999 ...
```

Note that `\rmntonum` will not check the numeral for formal validity. For example, both 'v' and 'vx' would yield '5'.

4 Revision history

1.2 2007-07-13

Added \DeclareListParser 3.8
Added \docsvlist 3.8
Added \rmntonum 3.9

1.1 2007-05-28

Added \protected@csedef 3.1
Added \protected@csxdef 3.1
Added \gluedef 3.2
Added \gluegdef 3.2
Added \csgluedef 3.2
Added \csgluegdef 3.2
Added \mudef 3.2
Added \mugdef 3.2
Added \csmudef 3.2
Added \csmugdef 3.2
Added \protected@eappto 3.4.1
Added \protected@xappto 3.4.1
Added \protected@cseappto 3.4.1
Added \protected@csxappto 3.4.1
Added \protected@epreto 3.4.2
Added \protected@xpreto 3.4.2
Added \protected@csepreto 3.4.2
Added \protected@csxpreto 3.4.2
Fixed bug in \newrobustcmd 2.1
Fixed bug in \renewrobustcmd 2.1
Fixed bug in \providerobustcmd 2.1

1.0 2007-05-07

Initial public release