The \texttt{ltcmdhooks} module\footnote{This file has version v1.0f dated 2021/10/20, \copyright{} L\TeX{} Project.}

Frank Mittelbach Phelype Oleinik

December 3, 2022

Contents

1 Introduction 1

2 Restrictions and Operational details 2
  2.1 Patching 2
    2.1.1 Timing 3
  2.2 Commands that look ahead 3

3 Package Author Interface 3

4 The Implementation 4
  4.1 Execution plan 4
  4.2 Variables 5
  4.3 Variants 6
  4.4 Patching or delaying 6
  4.5 Patching commands 8
    4.5.1 Patching by expansion and redefinition 9
    4.5.2 Patching by retokenization 15
  4.6 Messages 19

Index 20

1 Introduction

This file implements generic hooks for (arbitrary) commands. In theory every command \texttt{\langle name\rangle} offers now two associated hooks to which code can be added using \texttt{\AddToHook} or \texttt{\AddToHookNext}.\footnote{In practice this is not supported for all types of commands, see section 2.2 for the restrictions that apply and what happens if one tries to use this with commands for which this is not supported.} These are

\texttt{cmd/\langle name\rangle/before} This hook is executed at the very start of the command execution after its arguments (if any) are parsed. The hook \texttt{\langle code\rangle} is wrapped in the command inside a call to \texttt{\UseHook{cmd/(name)/before}}, so the arguments passed to the command are not available in the hook \texttt{\langle code\rangle}.
Add the hook \AddToHook{cmd/name/after}{<after code>}
then, after \begin{document} the definition of \texttt{\textbackslash foo} will be:
\renewcommand\foo[2]{\UseHook{cmd/foo/after}\texttt{Code #1 for #2!}}

In other words, the mechanism is similar to what \texttt{etoolbox} offers with \texttt{\pretocmd} and \texttt{\apptocmd} with the important differences

- that code can be prepended or appended (i.e., added to the hooks) even if the command itself is not defined, because the defining package has not yet been loaded
- and that by using the hook management interface it is now possible to define how the code chunks added in these places are ordered, if different packages want to add code at these points.

2 Restrictions and Operational details

Adding arbitrary material to commands is tricky because most of the time we do not know what the macro expects as arguments when expanding and \TeX doesn’t have a reliable way to see that, so some guesswork has to be employed.

2.1 Patching

The code here tries to find out if a command was defined with \texttt{\newcommand} or \texttt{\DeclareRobustCommand} or \texttt{\NewDocumentCommand}, and if so it \textit{assumes} that the argument specification of the command is as expected (which is not fail-proof, if someone redefines the internals of these commands in devious ways, but is a reasonable assumption).

If the command is one of the defined types, the code here does a sandboxed expansion of the command such that it can be redefined again exactly as before, but with the hook code added.

If however the command is not a known type (it was defined with \texttt{\def}, for example), then the code uses an approach similar to \texttt{etoolbox’s} \texttt{\patchcmd} to retokenize the command with the hook code in place. This procedure, however, is more likely to fail if the catcode settings are not the same as the ones at the time of command’s definition, so not always adding a hook to a command will work.
2.1.1 Timing

When \AddToHook (or its expl3 equivalent) is called with a generic \texttt{cmd} hook, say, \texttt{cmd/foo/before}, for the first time (that is, no code was added to that same hook before), in the preamble of a document, it will store a patch instruction for that command until \texttt{\begin{document}}, and only then all the commands which had hooks added will be patched in one go. That means that no command in the preamble will have hooks patched into them.

At \texttt{\begin{document}} all the delayed patches will be executed, and if the command doesn’t exist the code is still added to the hook, but it will not be executed. After \texttt{\begin{document}}, when \AddToHook is called with a generic \texttt{cmd} hook the first time, the command will be immediately patched to include the hook, and if it doesn’t exist or if it can’t be patched for any reason, an error is thrown; if \AddToHook was already used in the preamble no new patching is attempted.

This has the consequence that a command defined or redefined after \texttt{\begin{document}} only uses generic \texttt{cmd} hook code if \AddToHook is called for the first time after the definition is made, or if the command explicitly uses the generic hook in its definition by declaring it with \texttt{\NewHookPair} adding \texttt{\UseHook} as part of the code.\footnote{We might change this behavior in the main document slightly after gaining some usage experience.}

2.2 Commands that look ahead

Some commands are defined in different “steps” and they look ahead in the input stream to find more arguments. If you try to add some code to the \texttt{cmd/⟨name⟩/after} hook of such command, it will not work, and it is not possible to detect that programmatically, so the user has to know (or find out) which commands can or cannot have hooks attached to them.

One good example is the \texttt{\section} command. You can add something to the \texttt{cmd/section/before} hook, but if you try to add something to the \texttt{cmd/section/after} hook, \texttt{\section} will no longer work. That happens because the \texttt{\section} macro takes no argument, but instead calls a few internal \LaTeX{} macros to look for the optional and mandatory arguments. By adding code to the \texttt{cmd/section/after} hook, you get in the way of that scanning.

3 Package Author Interface

The \texttt{cmd} hooks are, by default, available for all commands that can be patched to add the hooks. For some commands, however, the very beginning or the very end of the code is not the best place to put the hooks, for example, if the command looks ahead for arguments (see section 2.2).

If you are a package author and you want to add the hooks to your own commands in the proper position you can define the command and manually add the \texttt{\UseHook} calls inside the command in the proper positions, and manually define the hooks with \texttt{\NewHook} or \texttt{\NewReversedHook}. When the hooks are explicitly defined, patching is not attempted so you can make sure your command works properly. For example, an (admittedly not really useful) command that typesets its contents in a framed box with width optionally given in parentheses:

\begin{verbatim}
\newcommand\fancybox{\Ifnextchar{{\fancybox}{\fancybox(5cm)}}
\def\fancybox(#1)#2{\fbox{\parbox{#1}{#2}}}}
\end{verbatim}
If you try that definition, then add some code after it with
\AddToHook{cmd/fancybox/after}{<code>}

and then use the \fancybox command you will see that it will be completely broken, because the hook will get executed in the middle of parsing for optional (...) argument.

If, on the other hand, you want to add hooks to your command you can do something like:
\newcommand\fancybox{%
  \UseHook{cmd/fancybox/before}\
  \parbox{#1}{#2}\
  \UseHook{cmd/fancybox/after}}
\NewHook{cmd/fancybox/before}
\NewReversedHook{cmd/fancybox/after}

then the hooks will be executed where they should and no patching will be attempted. It is important that the hooks are declared with \NewHook or \NewReversedHook, otherwise the command hook code will try to patch the command. Note also that the call to \UseHook{cmd/fancybox/before} does not need to be in the definition of \fancybox, but anywhere it makes sense to insert it (in this case in the internal @fancybox).

Alternatively, if for whatever reason your command does not support the generic hooks provided here, you can disable a hook with \DisableHook, so that when someone tries to add code to it they will get an error. Or if you don’t want the error, you can simply declare the hook with \NewHook and never use it.

The above approach is useful for really complex commands where for one or the other reason the hooks can’t be placed at the very beginning and end of the command body and some hand-crafting is needed. However, in the example above the real (and in fact only) issue is the cascading argument parsing in the style developed long ago in \LaTeX 2.09. Thus, a much simpler solution for this case is to replace it with the modern \NewDocumentCommand syntax and define the command as follows:
\DeclareDocumentCommand\fancybox{D(){5cm}m}{\fbox{\parbox{#1}{#2}}}

If you do that then both hooks automatically work and are patched into the right places.

4 The Implementation

4.1 Execution plan

To add before and after hooks to a command we will need to peek into the definition of a command, which is always a tricky thing to do. Some cases are easy because we know how the command was defined, so we can assume how its (parameter text) looks like (for example a command defined with \newcommand may have an optional argument followed by a run of mandatory arguments), so we can just expand that command and make it grab #1, #2, etc. as arguments and define it all back with the hooks added.

Life’s usually not that easy, so with some commands we can’t do that (a #1 might as well be #1212 instead of the expected #612, for example) so we need to resort to

3Please use \DisableHook if at all, only on hooks that you “own”, i.e., for commands that your package or class defines and not second guess whether or not hooks of other packages should get disabled!
“patching” the command: read its \texttt{meaning}, and tokenize it again with \texttt{scantokens} and hope for the best.

So the overall plan is:

1. Check if a command is of a known type (that is, defined with \texttt{newcommand}, \texttt{DeclareRobustCommand}, or \texttt{New(Expandable)DocumentCommand}), and if is, take appropriate action.

2. If the command is not a known type, we’ll check if the command can be patched. Two things will prevent a command from being patched: if it was defined in a nonstandard catcode setting, or if it is an internal expl3 command with \texttt{⟨⟨module⟩⟩} in its name, in which case we refuse to patch.

3. If the command was defined in nonstandard catcode settings, we will try a few standard ones to try our best to carry out the pathing. If this doesn’t help either, the code will give up and throw an error.

4. Variables

\texttt{\g_hook_patch_action_list_tl} Pairs of \texttt{\if<cmd>..\patch<cmd>} to be used with \texttt{\robust@command@act} when looking for a known patching rule. This token list is exposed because we see some future applications (with very specialized packages, such as \texttt{etoolbox} that may want to extend the pairs processed. It is not meant for general use which is why it is not documented in the interface documentation above.

\texttt{\l__hook_patch_num_args_int} The number of arguments in a macro being patched.

\texttt{\l__hook_patch_prefixes_tl, \l__hook_param_text_tl, and \l__hook_replace_text_tl} The prefixes and parameters of the definition for the macro being patched.

\texttt{\c__hook_hash_tl} A constant token list that contains two parameter tokens.

\texttt{\tl_const:Nn \c__hook_hash_tl { # # }}

\footnote{It’s not always possible to reliably detect this case because a command defined with no optional argument is indistinguishable from a \texttt{\defed} command.}
Two temporary macros that change depending on the macro being patched.
\begin{verbatim}
\cs_new_eq:NN \__hook_exp_not:NN ?
\cs_new_eq:NN \__hook_def_cmd:w ?
\end{verbatim}
(End definition for \__hook_exp_not:NN and \__hook_def_cmd:w.)

Internal quarks for recursion: they can’t appear in any macro being patched.
\begin{verbatim}
\quark_new:N \q__hook_recursion_tail
\quark_new:N \q__hook_recursion_stop
\end{verbatim}
(End definition for \q__hook_recursion_tail and \q__hook_recursion_stop.)

A list containing the patches delayed to \begin{document}, so that patching is not attempted twice.
\begin{verbatim}
\prop_new:N \g__hook_delayed_patches_prop
\end{verbatim}
(End definition for \g__hook_delayed_patches_prop.)

A helper for patching debug info.
\begin{verbatim}
\cs_new_protected:Npn \__hook_patch_debug:x #1
\cs_new_protected:Npn \__hook_patch_debug:w
\cs_new_protected:Npn \__hook_patch_debug:x #1
\end{verbatim}
(End definition for \__hook_patch_debug:x.)

4.3 Variants
\begin{verbatim}
\tl_rescan:nV \tl_rescan:nn
\cs_generate_variant:Nn \tl_rescan:nn { nV }
\end{verbatim}
(End definition for \tl_rescan:nV.)

4.4 Patching or delaying
Before \begin{document} all patching is delayed.
\begin{verbatim}
\__hook_try_put_cmd_hook:n \__hook_try_put_cmd_hook:w
\end{verbatim}
This function is called from within \AddToHook, when code is first added to a generic cmd hook. If it is called within the preamble, it delays the action until \begin{document}; otherwise it tries to update the hook.
\begin{verbatim}
\begin{verbatim}
\end{verbatim}
(End definition for \end{verbatim})

(lateXrelease)\IncludeInRelease{2021/11/15}{\__hook_try_put_cmd_hook:n}{\__hook_try_put_cmd_hook:w}
\begin{verbatim}
\end{verbatim}
(End definition for \end{verbatim})

(lateXrelease)\IncludeInRelease{2021/11/15}{\__hook_patch_cmd_or_delay:Nnn}{\__hook_patch_cmd_or_delay:Nnn}
\begin{verbatim}
\end{verbatim}
(End definition for \end{verbatim})
\IncludeInRelease{2021/06/01}{\__hook_try_put_cmd_hook:n}{Standardise\-generic\-hook\-names}
{ \__hook_try_put_cmd_hook:w #1 / / / \s__hook_mark {#1} }
\EndIncludeInRelease

\cs_new_protected:Npn \__hook_try_put_cmd_hook:w #1 #2 / #3 / #4 \s__hook_mark #5
\EndIncludeInRelease

\__hook_debug:n { \iow_term:n { ->~Adding\-cmd\-hook\-to\-'#2'\-(#3): } }
\EndIncludeInRelease

\__hook_patch_cmd_or_delay:Nnn \__hook_cmd_begindocument_code:
\__hook_patch_cmd_or_delay:Nnn just adds the patch instruction to a property list to be executed later.
\cs_gset_eq:NN \__hook_patch_cmd_or_delay:Nnn \__hook_patch_command:Nnn
\prop_map_function:NN \g__hook_delayed_patches_prop { \use_ii:nn }
\prop_gclear:N \g__hook_delayed_patches_prop
\cs_undefine:N \__hook_cmd_begindocument_code:
\EndIncludeInRelease

\__hook_cmd_try_patch:nn At \begin{document} tries patching the command if the hook was not manually created in the meantime. If the document does not exist, no error is raised here as it may hook into a package that wasn’t loaded. Hooks added to commands in the document body still raise an error if the command is not defined.
\cs_gset_eq:NN \__hook_patch_cmd_or_delay:Nnn \__hook_patch_command:Nnn
\prop_map_function:NN \g__hook_delayed_patches_prop { \use_ii:nn }
\prop_gclear:N \g__hook_delayed_patches_prop
\cs_undefine:N \__hook_cmd_begindocument_code:
\EndIncludeInRelease

\__hook_cmd_try_patch:nn
\__hook_patch_command:Nnn will do some sanity checks on the argument to detect if it is possible to add hooks to the command, and raises an error otherwise. If the command can contain hooks, then it uses \robust@command@act to find out what type is the command, and patch it accordingly.

\cs_new_protected:Npn \__hook_patch_command:Nnn #1 #2 #3
{ \__hook_patch_debug:x { analyzing: \token_to_str:N #1 } \__hook_patch_check:NNnn \cs_if_exist:NTF #1 { undef } { \__hook_patch_debug:x { ++ control sequence is defined } \__hook_patch_check:NNnn \token_if_macro:NTF #1 { macro } { \__hook_patch_debug:x { ++ control sequence is a macro } \__hook_patch_check:NNnn \__hook_if_public_command:NTF #1 { expl3 } { \__hook_patch_debug:x { ++ macro is not private } \robust@command@act \g_hook_patch_action_list_tl #1 \__hook_retokenize_patch:Nnn { #1 {#2} {#3} } } } }

And here's the auxiliary used above:
\cs_new_protected:Npn \__hook_patch_check:NNnn \__hook_if_public_command:N
{ \__hook_patch_check:NNnn \__hook_patch_check:NNnn \cs_if_exist:NTF \__hook_if_public_command:w
{ \msg_error:nnxx { hooks } { cant-patch } \token_to_str:N \__hook_patch_check:NNnn {#1} \__hook_patch_check:NNnn {#2} \__hook_patch_check:NNnn {#3} } and a conditional \__hook_if_public_command:N to check if a command has __ in its name (no other checking is performed). Primitives with :D in their name could be included here, but they are already discarded in the \token_if_macro:NTF test above.
4.5.1 Patching by expansion and redefinition

This is the list of known command types and the function that patches the command hooks into them. The conditionals are taken from `\ShowCommand`, `\NewCommandCopy` and `\__kernel_cmd_if_xparse:NTF` defined in ltcmd.

\tl_gset:Nn \g_hook_patch_action_list_tl
\{ \__hook_patch_DeclareRobustCommand:Nnn \}
\{ \__hook_patch_newcommand:Nnn \}
\{ \__kernel_cmd_if_xparse:NTF \__hook_cmd_patch_xparse:Nnn \}
\} (End definition for `\__hook_patch_action_list_tl`.)

\__hook_patch_DeclareRobustCommand:Nnn

At this point we know that the commands can be patched by expanding then redefining. These are the cases of commands defined with `\newcommand` with an optional argument or with `\DeclareRobustCommand`.

With `\__hook_patch_DeclareRobustCommand:Nnn` we check if the command has an optional argument (with a test counter-intuitively called `\@if@newcommand`; also make sure the command doesn’t take args by calling `\robust@command@chk@safe`). If so, we pass the patching action to `\__hook_patch_newcommand:Nnn` and otherwise we call the patching engine `\__hook_patch_expand_redefine:Nnn` with a `\c_false_bool` to indicate that there is no optional argument.
\__hook_patch_newcommand:Nnn

If the command was defined with \newcommand and an optional argument, call the patching engine with a \c_true_bool to flag the presence of an optional argument, and with \command to patch the actual code for \command.

\cs_new_protected:Npn \__hook_patch_newcommand:Nnn #1
\exp_args:NNc \__hook_patch_expand_redefine:NNnn \c_true_bool
\{ \cs_to_str:N #1 \}

(End definition for \__hook_patch_newcommand:Nnn.)

\__hook_cmd_patch_xparse:Nnn

And for commands defined by the xparse commands use this for patching:

\cs_new_protected:Npn \__hook_cmd_patch_xparse:Nnn #1
\exp_args:NNc \__hook_patch_expand_redefine:NNnn \c_false_bool
\{ \cs_to_str:N #1 \_code \}

(End definition for \__hook_cmd_patch_xparse:Nnn.)

Now the real action begins. Here we have in #1 a boolean indicating if the command has a leading [...] delimited argument, in #2 the command control sequence, in #3 the name of the command (note that #1 \neq \csname#2\endcsname at this point!), and in #4 the hook position, either before or after.

Patching with expansion+redefinition is trickier than it looks like at first glance. Suppose the simple definition:

\def\foo#1{#1##2}

When defined, its (replacement text) will be a token list containing:

\texttt{out\_param \ 1, mac\_param \#, character \ 2}

Then, after expanding \foo{##1} (here ## denotes a single #6) we end up with a token list with out\_param \ 1 replaced:

\texttt{mac\_param \#, character \ 1, mac\_param \#, character \ 2}

that is, the definition would be:

\def\foo#1{#1##2}

which obviously fails, because the original input in the definition was ## but \TeX{} reduced that to a single parameter token #6 when carrying out the definition. That leaves no room for a clever solution with (say) unexpanded, because anything that would double the second #6 would also (incorrectly) double the first, so there's not much to do other than a manual solution.

There are three cases we can distinguish to make things hopefully faster on simpler cases:
1. a macro with no parameters;
2. a macro with no parameter tokens in its definition;
3. a macro with parameters and parameter tokens.

The first case is trivial: if the macro has no parameters, we can just use \texttt{\unexpanded} around it, and if there is a parameter token in it, it is handled correctly (the macro can be treated as a \texttt{tl} variable).

The second case requires looking at the \texttt{replacement text} of the macro to see if it has a parameter token in there. If it does not, then there is no worry, and the macro can be redefined normally (without \texttt{\unexpanded}).

The third case, as usual, is the devious one. Here we’ll have to loop through the definition token by token, and double every parameter token, so that this case can be handled like the previous one.

\begin{verbatim}
cs_new_protected:Npn \__hook_patch_expand_redefine:NNNn #1 #2 #3 #4
{ \__hook_patch_debug:x { ++~command~can~be~patched~without~rescanning }

\int_set:Nn \l__hook_patch_num_args_int
\exp_args:Nf \str_count:n { \cs_argument_spec:N #2 } / 2
\bool_if:NT #1 { -1 }
}
\end{verbatim}

We’ll start by counting the number of arguments in the command by counting the number of characters in the \texttt{\cs_argument_spec:N} of the macro, divided by two, and subtracting one if the command has an optional argument (that is, an extra \texttt{[\ldots]} in its \texttt{replacement text}).

\begin{verbatim}
\tl_set:Nx \l__hook_tmpa_tl { \bool_if:NTF #1 { [ ] } { { } } }
\int_step_inline:nnn { 2 } { \l__hook_patch_num_args_int }
\exp_args:NNo \exp_args:No \__hook_if_has_hash:nTF
{ \exp_after:wN #2 \l__hook_tmpa_tl }
{ \cs_set_eq:NN \__hook_exp_not:n \exp_not:n }
\end{verbatim}

Now build two token lists:

\texttt{\l__hook_param_text_tl} will contain the \texttt{replacement text} to be used when redefining the macro. It should be identical to the \texttt{replacement text} used when originally defining that macro.

\texttt{\l__hook_replace_text_tl} will contain braced pairs of \texttt{\c__hook_hash_tl\texttt{\langle num\rangle}} to feed to the macro when expanded. This token list as well as the previous will have the first item surrounded by \texttt{[\ldots]} in the case of an optional argument.

The use of \texttt{\c__hook_hash_tl} here is to differentiate actual parameters in the macro from parameter tokens in the original definition of the macro. Later on, \texttt{\c__hook_hash_tl} is either replaced by actual parameter tokens, or expanded into them.

\begin{verbatim}
\int_compare:nNnTF { \l__hook_patch_num_args_int } > { \c_zero_int }
{ \begin{verbatim}
\tl_set:Nx \l__hook_tempa_tl \{ \bool_if:NTF #1 { [ ] } { { } } \}
\int_step_inline:nnn { 2 } { \l__hook_patch_num_args_int }
\tl_put_right:Nn \l__hook_tempa_tl \{ \}
\exp_args:NNo \exp_args:No \__hook_if_has_hash:nTF
{ \exp_after:wN #2 \l__hook_tempa_tl }
{ \cs_set_eq:NN \__hook_exp_not:n \exp_not:n }
\end{verbatim}
\end{verbatim}

We’ll first check if the command has any parameter token in its definition (feeding it empty arguments), and set \texttt{\__hook_exp_not:n} accordingly. \texttt{\__hook_exp_not:n} will be used later to either leave \texttt{\c__hook_hash_tl} or expand it, and also to remember the result of \texttt{\__hook_if_has_hash:nTF} to avoid testing twice (the test can be rather slow).
Here we’ll conditionally add \[ \ldots \] around the first parameter:
\bool_if:NTF #1
\{ \__hook_tmp:w \tl_set:Nx { \[ \c__hook_hash_tl 1 \] } \}
\{ \__hook_tmp:w \tl_set:Nx { \{ \c__hook_hash_tl 1 \} } \}

Then, for every parameter from the second, just add it normally:
\int_step_inline:nnn { 2 } { \l__hook_patch_num_args_int }
\{ \__hook_tmp:w \tl_put_right:Nx { \{ \c__hook_hash_tl ##1 \} } \}

Now, if the command has any parameter token in its definition (then \__hook_exp_not:n is \exp_not:n), call \__hook_double_hashes:n to double them, and replace every \c__-hook_hash_tl by #:
\tl_set:Nx \l__hook_replace_text_tl
\exp_not:N #2 \exp_not:V \l__hook_replace_text_tl
\tl_set:Nx \l__hook_replace_text_tl
\token_if_eq_meaning:NNTF \__hook_exp_not:n \exp_not:n
\exp_args:NNV \exp_args:No \__hook_double_hashes:n
\exp_args:NV \exp_not:o
\l__hook_replace_text_tl

And now, set a few auxiliaries for the case that the macro has parameters, so it won’t be passed through \unexpanded (twice):
\cs_set_eq:NN \__hook_def_cmd:w \tex_gdef:D
\cs_set_eq:NN \__hook_exp_not:NN \prg_do_nothing:
\cs_set:Npn \__hook_exp_not:NN ##1 { \exp_not:N ##1 \exp_not:N }

In the case the macro has no parameters, we’ll treat it as a token list and things are much simpler (expansion control looks a bit complicated, but it’s just a pair of \exp_not:N preventing another \exp_not:n from expanding):
\tl_set:Nx \l__hook_param_text_tl
\tl_set_eq:NN \l__hook_replace_text_tl \#2
\cs_set_eq:NN \__hook_def_cmd:w \tex_xdef:D
\cs_set:Npn \__hook_exp_not:NN #1 { \exp_not:N #1 \exp_not:N }
Finally, call \__hook_redefine_with_hooks:Nnnn with the macro being redefined in \#1, then \UseHook{cmd/<name>/before} in \#2 or \UseHook{cmd/<name>/after} in \#3 (one is always empty), and in \#4 the (replacement text) of the macro.

\use:x
{ \__hook_redefine_with_hooks:Nnnn \exp_not:N #2
  \str_if_eq:nnTF {#4} { after } { \use:nn } { \use_ii_i:nn }
  { \__hook_exp_not:NN \exp_not:N \UseHook { cmd / #3 / #4 } } }
{ \__hook_exp_not:NN \exp_not:V \l__hook_replace_text_tl }
}\use:x
\__hook_redefine_with_hooks:Nnnn #1 #2 #3 #4
{ \l__hook_patch_prefixes_tl \exp_after:wN \__hook_def_cmd:w \exp_after:wN #1 \l__hook_param_text_tl { #2 #4 #3 } }
{ \l__hook_patch_prefixes_tl \exp_after:wN \__hook_def_cmd:w \exp_after:wN \#1 \l__hook_param_text_tl }
{ \#2 \#4 \#3 }

Here’s the auxiliary that makes the prefix control sequences for the redefinition. Each item has to be \tl_trim_spaces:n’d because the last item (and not any other) has a trailing space.
\cs_new:Npn \__hook_make_prefixes:w / #1 /
{ \tl_if_empty:nF {#1} { \exp_not:c { tex_ \tl_trim_spaces:n {#1} :D } \__hook_make_prefixes:w / #1 / } }
{ \__hook_make_prefixes:w / }
(End definition for \__hook_patch_expand_redefine:Nnnn, \__hook_redefine_with_hooks:Nnnn, and \__hook_make_prefixes:w.)

Here are some auxiliaries for the contraption above.
\__hook_if_has_hash_p:n \__hook_if_has_hash:nTF \__hook_if_has_hash:w \__hook_if_has_hash_check:w
\__hook_if_has_hash:nTF searches the token list \#1 for a catcode 6 token, and if any is found, it returns true, and false otherwise. The searching doesn’t care about preserving groups or spaces: we can ignore those safely (braces are removed) so that searching is as fast as possible.
\__hook_double_hashes:n loops through the token list \#1 and duplicates any catcode 6
token, and expands tokens \ifx-equal to \c__hook_hash_tl, and leaves all other tokens
\notexpanded with \exp_not:N. Unfortunately pairs of explicit catcode 1 and catcode 2
character tokens are normalised to \{1 and }1 because it’s not feasible to expandably
detect the character code (maybe it could be done using something along the lines of
https://tex.stackexchange.com/a/527538, but it’s far too much work for close to
zero benefit).

\__hook_double_hashes:w is the tail-recursive loop macro, that tests which of the
three types of item is in the head of the token list.

\__hook_double_hashes:n checks for the end of the token list, then checks
if the token is \c__hook_hash_tl, and if so just leaves it.

(this \use_i:nnnn uses \fi: and consumes \use:n, the whole \if_catcode:w block, and the \exp_not:N, leaving just \#1 which is \c__hook_hash_tl.)
If \#1 is not \c__hook_hash_tl, then check if its catcode is 6, and if so, leave it doubled in \texttt{\texttt{\exp_not:n}} and consume the following \texttt{\exp_not:N} \#1.

\begin{verbatim}
\if_catcode:w ## \exp_not:N #1
\exp_after:wN \use_ii:nnnn
\fi:
\use_none:n
{ \exp_not:n { #1 #1 } }
\end{verbatim}

If both previous tests returned false, then leave the token unexpanded and resume the loop.

\begin{verbatim}
\exp_not:N #1
\__hook_double_hashes:w
\cs_new:Npn \__hook_double_hashes_stop:w #1 \q__hook_recursion_stop { \fi: }
\end{verbatim}

Dealing with spaces and grouped tokens is trivial:

\begin{verbatim}
\cs_new:Npn \__hook_double_hashes_group:n #1
{ { \__hook_double_hashes:n {#1} } \__hook_double_hashes:w }
\exp_last_unbraced:NNo
\cs_new:Npn \__hook_double_hashes_space:w \c_space_tl
{ ~ \__hook_double_hashes:w }
\end{verbatim}

(End definition for \texttt{\__hook_double_hashes:n} and others.)

\subsection*{4.5.2 Patching by retokenization}

At this point we've drained the possibilities of patching a command by expansion-and-redefinition, so we have to resort to patching by retokenizing the command. Patching by retokenization is done by getting the \texttt{\meaning} of the command, doing the necessary manipulations on the generated string, and the retokenizing that again by using \texttt{\scantokens}.

Patching by retokenization is definitely a riskier business, because it relies that the tokens printed by \texttt{\meaning} produce the exact same tokens as the ones in the original definition. That is, the catcode régime must be exactly(ish) the same, and there is no way of telling except by trial and error.

This is the macro that will control the whole process. First we'll try out one final, rather trivial case, of a command with no arguments; that is, a token list. This case can be patched with the expand-and-redefine routine but it has to be the very last case tested for, because most (all?) robust commands start with a top-level macro with no arguments, so testing this first would short-circuit \texttt{\robust@command@act} and the top-level macros would be incorrectly patched. In that case, we just check if the \texttt{\cs_argument_spec:N} is empty, and call \texttt{\__hook_patch_expand_redefine:Nnnn}.

\begin{verbatim}
\cs_new_protected:Npn \__hook_retokenize_patch:Nnn #1 #2 #3
{ \__hook_patch_debug:x { ..command-can-only-be-patched-by-rescanning } \str_if_eq:eeTF { \cs_argument_spec:N #1 } { } { \__hook_patch_expand_redefine:Nnnn \c_false_bool #1 {#2} {#3} } 
{ }

Otherwise, we start the actual patching by retokenization job. The code calls \texttt{\__hook_try_patch_with_catcodes:Nnnnnw} with a different catcode setting:
• The current catcode setting;
• Switching the catcode of \@;
• Switching the expl3 syntax on or off;
• Both of the above.

If patching succeeds, \__hook_try_patch_with_catcodes:Nnnnw has the side-effect of patching the macro #1 (which may be an internal from the command whose name is #2).

\tl_set:Nx \l__hook_tmpa_tl
\fill
\int_compare:nNnTF { \char_value_catcode:n {'\@ } } = { 12 }
{ \exp_not:N \makeatletter } { \exp_not:N \makeatother }
\tl_set:Nx \l__hook_tmpb_tl
\bool_if:NTF \l__kernel_expl_bool
{ \ExplSyntaxOff } { \ExplSyntaxOn }
\use:x
{ \exp_not:N \__hook_try_patch_with_catcodes:Nnnnw
 \exp_not:n { #1 {#2} {#3} }
{ \prg_do_nothing: }% @
\exp_not:V \l__hook_tmpa_tl % @
\exp_not:V \l__hook_tmpb_tl % _:
}
\q_recursion_tail \q_recursion_stop

\msg_error:nnxx { hooks } { cant-patch }
{ \c_backslash_str #2 } { retok }
\end}

(End definition for \__hook_retokenize_patch:Nnn.)

\__hook_try_patch_with_catcodes:Nnnw
This function is a simple wrapper around \__hook_cmd_if_scanable:NnTF and \__hook_-
patch_retokenize:Nnnn if the former returns \true, plus some debug messages.
\cs_new_protected:Npn \__hook_try_patch_with_catcodes:Nnnw #1 #2 #3 #4
{ \quark_if_recursion_tail_stop_do:nn {#4} { \use:n }
\__hook_patch_debug:x { ++-trying-to-patch-by-retokenization }
\__hook_cmd_if_scanable:NnTF {#1} {#4}
{ \__hook_patch_debug:x { ++-macro-can-be-retokenized-cleanly }
\__hook_patch_debug:x { ==--retokenizing-macro-now }
This is an oddity required to be safe (as safe as reasonably possible) when patching the command. The entirety of

\( \text{⟨prefixes⟩ \texttt{\textbackslash def ⟨cs⟩ ⟨parameter text⟩ ⟨replacement text⟩}} \)

will go through \texttt{\textbackslash scantokens}. The ⟨parameter text⟩ and ⟨replacement text⟩ are what we are trying to retokenize, so not much worry there. The other items, however, should “just work”, so some care is needed to not use too fancy catcode settings. Therefore we can’t use an expl3-named macro for ⟨cs⟩, nor the expl3 versions of \texttt{\textbackslash def} or the ⟨prefixes⟩. That is why the definitions that will eventually go into \texttt{\textbackslash scantokens} will use the oddly (but hopefully clearly)-named \texttt{\kerneltmpDoNotUse}:

\texttt{\textbackslash cs_new_eq:NN \kerneltmpDoNotUse !}

\textit{PhO:} Maybe this can be avoided by running the ⟨parameter text⟩ and ⟨replacement text⟩ separately through \texttt{\textbackslash scantokens} and then putting everything together at the end.

(End definition for \texttt{\kerneltmpDoNotUse}.)

\texttt{\_hook_patch_required_catcodes:}

Here are the catcode settings that are \textit{mandatory} when retokenizing commands. These are the minimum necessary settings to perform the definitions: they identify control sequences, which must be escaped with \texttt{\textbackslash 0}, delimit the definition with \{1 and \}2, and mark parameters with \texttt{#6}. Everything else may be changed, but not these.

\texttt{\textbackslash cs_new_protected:Npn \_\_hook_patch_required_catcodes:}

\{
  \char_set_catcode_escape:N \textbackslash \\
  \char_set_catcode_group_begin:N \{
  \char_set_catcode_group_end:N \}
  \char_set_catcode_parameter:N \#
  \% \texttt{\textbackslash int_set:Nn \texttt{\textbackslash tex_endlinechar:D \{-1 \}}}
  \% \texttt{\textbackslash int_set:Nn \texttt{\textbackslash tex_newlinechar:D \{-1 \}}}
\}

\textit{PhO:} etoolbox sets the \texttt{\endlinechar} and \texttt{\newlinechar} when patching, but as far as I tested these didn’t make much of a difference, so I left them out for now. Maybe \texttt{\newlinechar=-1} avoids a space token being added after the definition.

\textit{PhO:} If the patching is split by ⟨parameter text⟩ and ⟨replacement text⟩, then only \# will have to stay in that list.

\textit{PhO:} Actually now that we patch \texttt{\textbackslash UseHook(cmd/foo/before)}, all the tokens there need to have the right catcodes, so this list now includes all lowercase letters, \texttt{U} and \texttt{H}, the slash, and whatever characters in the command name... sigh...

(End definition for \texttt{\_\_hook_patch_required_catcodes:}.)

\kerneltmpDoNotUse
Here we'll do a quick test if the command being patched can in fact be retokenized with the specific catcode setting without changing in meaning. The test is straightforward:

1. apply \meaning to the command;
2. split the ⟨prefixes⟩, ⟨parameter text⟩ and ⟨replacement text⟩ and arrange them as
   ⟨prefixes⟩\def\kerneltmpDoNotUse⟨parameter text⟩{⟨replacement text⟩}
3. rescan that with the given catcode settings, and do the definition; then finally
4. compare \kerneltmpDoNotUse with the original command.

If both are \ifx-equal, the command can be safely patched.

\begin{verbatim}
\prg_new_protected_conditional:Npnn \__hook_cmd_if_scanable:Nn #1 #2 { TF }
  { \cs_set_eq:NN \kerneltmpDoNotUse \scan_stop:
    \cs_set_eq:NN \__hook_tmp:w \scan_stop:
    \use:x
      \cs_set:Npn \__hook_tmp:w
        ####1 \tl_to_str:n { macro: } ####2 -> ####3 \s__hook_mark
        { ####1 \def \kerneltmpDoNotUse ####2 {####3} }
    \tl_set:Nx \exp_not:N \l__hook_tmpa_tl
      { \exp_not:N \__hook_tmp:w \token_to_meaning:N #1 \s__hook_mark }
  \tl_rescan:nV { #2 \__hook_patch_required_catcodes: } \l__hook_tmpa_tl
  \token_if_eq_meaning:NNTF #1 \kerneltmpDoNotUse
    { \prg_return_true: } \prg_return_false: }
\end{verbatim}

(End definition for \__hook_cmd_if_scanable:NnTF.)

Then, if \__hook_cmd_if_scanable:NnTF returned true, we can go on and patch the command.

\begin{verbatim}
\cs_new_protected:Npn \__hook_patch_retokenize:Nnnn #1 #2 #3 #4
  { \cs_set_eq:NN \kerneltmpDoNotUse \scan_stop:
    \cs_set_eq:NN \__hook_tmp:w \scan_stop:
    \use:x
      \cs_set:Npn \__hook_tmp:w
        ####1 \tl_to_str:n { macro: } ####2 -> ####3 \s__hook_mark
        { ####1 \def \kerneltmpDoNotUse ####2 {####3} }
    \tl_rescan:nV { #2 \__hook_patch_required_catcodes: } \l__hook_tmpa_tl
    \token_if_eq_meaning:NNTF #1 \kerneltmpDoNotUse
      { \prg_return_true: } \prg_return_false: }
\end{verbatim}

(End definition for \__hook_patch_retokenize:Nnnn.)

Now we'll define \__hook_tmp:w such that it splits the \meaning of the macro (#1) into its three parts:

####1. ⟨prefixes⟩
####2. ⟨parameter text⟩
####3. ⟨replacement text⟩
and arrange that a complete definition, then place the \texttt{before} or \texttt{after} hooks around the \texttt{replacement text} accordingly.

\vspace{1em}
\begin{verbatim}
\cs_set:Npn \_hook_tmp:w
  ####1 \tl_to_str:n { macro: } ####2 -> ####3 \s__hook_mark
  { ####1 \def \kerneltmpDoNotUse ####2
  \str_if_eq:nTF {#3} { before } { \token_to_str:N \UseHook \cmd / #2 / #3 } ####3
  \str_if_eq:nTF {#3} { after } { \token_to_str:N \UseHook \cmd / #2 / #3 } ####3
  }
\end{verbatim}

Now we just have to get the \texttt{meaning} of the command being patched and pass it through the meat grinder above.

\vspace{1em}
\begin{verbatim}
\tl_set:Nx \exp_not:N \l__hook_tmpa_tl
\exp_not:N \__hook_tmp:w \token_to_meaning:N #1 \s__hook_mark
\end{verbatim}

Now rescan with the given catcode settings (overridden by the \_\_hook_patch_-\_required_catcodes\,), and implicitly (by using the rescanned token list) carry out the definition from above.

\begin{verbatim}
\tl_rescan:nV { #4 \_\_hook_patch_required_catcodes: } \l__hook_tmpa_tl
\end{verbatim}

And to close, copy the newly-defined command into the old name and the patching is finally completed:

\vspace{1em}
\begin{verbatim}
\cs_gset_eq:NN #1 \kerneltmpDoNotUse
\end{verbatim}

(End definition for \_\_hook_patch_retokenize:Nnnn.)

4.6 Messages

\begin{verbatim}
\msg_new:nnnn { hooks } { wrong-cmd-hook }
\msg_new:nnnn { hooks } { cant-patch }
\end{verbatim}
You tried to add a hook to \texttt{\#1}, but LaTeX was unable to patch the command because it \texttt{\_\_hook_unpatchable_cases:n} \texttt{(\#2)}.

\cs_new:Npn \_\_hook_unpatchable_cases:n {\#1}
\{
  \str_case:nn {\#1}
  {
    { undef } { doesn't exist }
    { macro } { is not a macro }
    { expl3 } { is a private expl3 macro }
    { retok } { can't be retokenized cleanly }
  }
\}

\IncludeInRelease{0000/00/00}{ltcmdhooks}{The hook management system for commands}
\EndModuleRelease
\ExplSyntaxOff

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols

\# ........................................ 341
\/ ........................................ 199
\\ .................................... 338, 397
\{ ......................................... 339
\} ......................................... 340
A
\AddToHook .................................... 3
B
\begin ........................................ 65
bool commands:
\bool_if:NTF ........ 159, 163, 175, 297
\c_false_bool ........ 140, 150, 288, 9
\c_true_bool ........... 145, 10
C
char commands:
\char_set_catcode_escape:N .... 338
\char_set_catcode_group_begin:N 339
\char_set_catcode_group_end:N ... 340
\char_set_catcode_parameter:N ... 341
\char_value_catcode:n ............ 292
cs commands:
\cs_argument_spec:N ............ 158, 287, 15
\cs_generate_variant:Nn ........ 19
\cs_gset_eq:NN ................. 55, 384
\cs_if_exist:NTF ............. 72, 80
\cs_new:Npn ........... 224, 234,
244, 246, 248, 259, 278, 279, 282, 409
\cs_new_eq:NN ........... 335, 12, 13
\cs_new_protected:Npn .......... 53, 62, 76, 96, 116,
129, 134, 143, 148, 153, 217, 284,
319, 336, 362, 17, 22, 24, 33, 35, 45
\cs_prefix_spec:N ........... 204
\cs_set:Npn ............. 196, 351, 368
\texttt{\textbackslash T}{\textbackslash H} and \texttt{\textbackslash L}{\textbackslash H}E\TeX and \LaTeX\ commands:
\begin{itemize}
\item \texttt{\textbackslash def} 17
\item \texttt{\textbackslash endlinechar} 17
\item \texttt{\textbackslash escapechar} 12
\item \texttt{\textbackslash g@addto@macro} 60
\item \texttt{\textbackslash ifx} 18
\item \texttt{\textbackslash meaning} 15
\item \texttt{\textbackslash newlinechar} 17
\item \texttt{\textbackslash noexpand} 18
\item \texttt{\textbackslash patchcmd} 2
\item \texttt{\textbackslash relax} 18
\item \texttt{\textbackslash robust@command@act} 89, 5
\item \texttt{\textbackslash robust@command@chk@safe} 136, 9
\item \texttt{\textbackslash scantokens} 5
\item \texttt{\textbackslash section} 3
\item \texttt{\textbackslash ShowCommand} 9
\item \texttt{\textbackslash \textbackslash tex} commands:
\begin{itemize}
\item \texttt{\textbackslash tex_endlinechar:D} 342
\item \texttt{\textbackslash tex_escapechar:D} 199
\item \texttt{\textbackslash tex_gdef:D} 189, 13
\item \texttt{\textbackslash tex_neulinechar:D} 343
\item \texttt{\textbackslash tex_xdef:D} 195, 13
\end{itemize}
\item \texttt{\textbackslash tl} commands:
\begin{itemize}
\item \texttt{\textbackslash c\_space\_tl} 282
\end{itemize}
\end{itemize}

\texttt{\textbackslash \textbackslash use} commands:
\begin{itemize}
\item \texttt{\textbackslash use:n} 104, 169, 172, 200, 206, 267, 300, 321, 349, 366, 14
\item \texttt{\textbackslash use:nn} 211
\item \texttt{\textbackslash use_i:nnn} 265, 14
\item \texttt{\textbackslash use_i:delimit_by_q_recursion_-\textbackslash stop:nw} 328
\item \texttt{\textbackslash use_i:i:nn} 56, 138
\item \texttt{\textbackslash use_i:i:nnnn} 270
\item \texttt{\textbackslash use_i:i:i:nn} 210
\item \texttt{\textbackslash use_i:i:i:i:nnn} 272, 328
\end{itemize}

\item \texttt{\textbackslash UseHook} 212, 374, 377, 3

\begin{itemize}
\item \texttt{\textbackslash tl_clear:N} 193
\item \texttt{\textbackslash tl_const:Nn} 11
\item \texttt{\textbackslash tl_gset:Nn} 123
\item \texttt{\textbackslash tl_if_empty:nTF} 119, 226, 245
\item \texttt{\textbackslash tl_if_head_is_group:nTF} 253
\item \texttt{\textbackslash tl_if_head_is_N_type:nTF} 250
\item \texttt{\textbackslash tl_new:N} 6, 8, 9, 10
\item \texttt{\textbackslash tl_put_right:Nn} 165, 179
\item \texttt{\textbackslash tl_rescan:nn} 357, 383, 19, 19
\item \texttt{\textbackslash tl_set:Nn} 163, 176, 177, 180, 182, 203, 290, 295, 354, 380
\item \texttt{\textbackslash tl_set_eq:NN} 194
\item \texttt{\textbackslash tl_to_str:n} 112, 117, 352, 369
\item \texttt{\textbackslash tl_trim_spaces:n} 228, 13
\item \texttt{\textbackslash token_if_eq:catcode:NNTF} 238
\item \texttt{\textbackslash token_if_eq_meaning:NNTF} 184, 358
\item \texttt{\textbackslash token_if_macro:NNTF} 83, 8
\item \texttt{\textbackslash token_to_meaning:N} 79, 355, 381
\item \texttt{\textbackslash token_to_str:N} 78, 79, 101, 374, 377
\item \texttt{\textbackslash \textbackslash U} commands:
\begin{itemize}
\item \texttt{\textbackslash unexpanded} 11
\end{itemize}

\end{itemize}