The `ltcmdhooks` module*

Frank Mittelbach Phelype Oleinik

May 4, 2021

Contents

1 Introduction

This file implements generic hooks for (arbitrary) commands. In theory every command \⟨*name*⟩ offers now two associated hooks to which code can be added using `\AddToHook` or `\AddToHookNext`.[1] These are

`cmd/`⟨*name*⟩`/before` This hook is executed at the very start of the command execution after its arguments (if any) are parsed. The hook ⟨*code*⟩ is wrapped in the command inside a call to `\UseHook{cmd/`⟨*name*⟩`/before}`, so the arguments passed to the command are *not* available in the hook ⟨*code*⟩.

*This file has version v1.0a dated 2021/04/30, © LaTeX Project.

[1] 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.

cmd/⟨*name*⟩/after This hook is similar to cmd/⟨*name*⟩/before, but it is executed at the very end of the command body. This hook is implemented as a reversed hook.

The hooks are not physically present before \begin{document} (i.e., using a command in the preamble will never execute them) and if nobody has declared any code for them, then they are not added to the command code ever. For example, if we have the following definition

```
\newcommand\foo[2]{Code #1 for #2!}
```

then executing \foo{A}{B} will simply run Code␣A␣for␣B! as it was always the case. However, if somebody, somewhere (e.g., in a package) adds

```
\AddToHook{cmd/foo/before}{<before code>}
```

then, after \begin{document} the definition of \foo will be:

```
\renewcommand\foo[2]{\UseHook{cmd/foo/before}Code #1 for #2!}
```

and similarly \AddToHook{cmd/foo/after}{<after code>} alters the definition to

```
\renewcommand\foo[2]{Code #1 for #2!\UseHook{cmd/foo/after}}
```

In other words, the mechanism is similar to what etoolbox offers with \pretocmd and \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 \newcommand or \DeclareRobustCommand or \NewDocumentCommand, and if so it *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 \def, for example), then the code uses an approach similar to etoolbox's \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 `cmd` hook, say, `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 `\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 `\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 `\begin{document}`, when `\AddToHook` is called with a generic `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 `\begin{document}` only uses generic `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 `\NewHookPair` adding `\UseHook` as part of the code.[2]

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 `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 `\section` command. You can add something to the `cmd/section/before` hook, but if you try to add something to the `cmd/section/after` hook, `\section` will no longer work. That happens because the `\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 `cmd/section/after` hook, you get in the way of that scanning.

3 Package Author Interface

The `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 `\UseHook` calls inside the command in the proper positions, and manually define the hooks with `\NewHook` or `\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:

```
\newcommand\fancybox{\@ifnextchar({\@fancybox}{\@fancybox(5cm)}}
\def\@fancybox(#1)#2{\fbox{\parbox{#1}{#2}}}
```

[2] We might change this behavior in the main document slightly after gaining some usage experience.

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{\@ifnextchar({\@fancybox}{\@fancybox(5cm)}}
\def\@fancybox(#1)#2{\fbox{%
                      \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`[3], 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 $\texttt{\#}_{12}1_{12}$ instead of the expected $\texttt{\#}_6 1_{12}$, for example) so we need to resort to

[3] Please 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 `\meaning`, and tokenize it again with `\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 `\newcommand`[4], `\DeclareRobustCommand`, or `\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 `__`⟨*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.

```
⟨@@=hook⟩
⟨*2ekernel | latexrelease⟩
\ExplSyntaxOn
⟨latexrelease⟩\NewModuleRelease{2021/06/01}{ltcmdhooks}
⟨latexrelease⟩                        {The~hook~management~system~for~commands}
```

4.2 Variables

`\g_hook_patch_action_list_tl`

Pairs of `\if<cmd>..\patch<cmd>` to be used with `\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 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.

```
\tl_new:N \g_hook_patch_action_list_tl
```

(*End definition for* `\g_hook_patch_action_list_tl`.)

`\l__hook_patch_num_args_int`

The number of arguments in a macro being patched.

```
\int_new:N \l__hook_patch_num_args_int
```

(*End definition for* `\l__hook_patch_num_args_int`.)

`\l__hook_patch_prefixes_tl`
`\l__hook_patch_param_text_tl`
`\l__hook_patch_replacement_tl`

The prefixes and parameters of the definition for the macro being patched.

```
\tl_new:N \l__hook_patch_prefixes_tl
\tl_new:N \l__hook_patch_param_text_tl
\tl_new:N \l__hook_patch_replacement_tl
```

(*End definition for* `\l__hook_patch_prefixes_tl`, `\l__hook_patch_param_text_tl`, *and* `\l__hook_patch_replacement_tl`.)

`\g__hook_delayed_patches_prop`

A list containing the patches delayed to `\begin{document}`, so that patching is not attempted twice.

```
\prop_new:N \g__hook_delayed_patches_prop
```

(*End definition for* `\g__hook_delayed_patches_prop`.)

[4] It's not always possible to reliably detect this case because a command defined with no optional argument is indistinguishable from a `\def`ed command.

__hook_patch_debug:x A helper for patching debug info.

```
\cs_new_protected:Npn \__hook_patch_debug:x #1
  { \__hook_debug:n { \iow_term:x { [lthooks]~#1 } } }
```

(*End definition for* `\__hook_patch_debug:x`.)

4.3 Variants

\tl_rescan:nV expl3 function variants used throughout the code.

```
\cs_generate_variant:Nn \tl_rescan:nn { nV }
```

(*End definition for* `\tl_rescan:nV`.)

4.4 Patching or delaying

__hook_try_put_cmd_hook:n
__hook_try_put_cmd_hook:w
Before `\begin{document}` all patching is delayed. This function is called from within `\AddToHook`, when code is added to a generic `cmd` hook is newly declared. It checks whether the patch position is valid, then proceeds to trying to patch or delaying to `\begin{document}` if in the preamble.

```
\cs_new_protected:Npn \__hook_try_put_cmd_hook:n #1
  { \__hook_try_put_cmd_hook:w #1 / / / \s__hook_mark {#1} }
\cs_new_protected:Npn \__hook_try_put_cmd_hook:w
    #1 / #2 / #3 / #4 \s__hook_mark #5
  {
    \__hook_debug:n { \iow_term:n { ->~Adding~cmd~hook~to~'#2'~(#3): } }
    \str_case:nnTF {#3}
        { { before } { } { after } { } }
      { \exp_args:Nc \__hook_patch_cmd_or_delay:Nnn {#2} {#2} {#3} }
      { \__kernel_msg_error:nnn { hooks } { wrong-cmd-hook } {#2} {#3} }
  }
```

(*End definition for* `\__hook_try_put_cmd_hook:n` *and* `\__hook_try_put_cmd_hook:w`.)

__hook_patch_cmd_or_delay:Nnn
__hook_cmd_begindocument_code:
In the preamble, `\__hook_patch_cmd_or_delay:Nnn` just adds the patch instruction to a property list to be executed later.

```
\cs_new_protected:Npn \__hook_patch_cmd_or_delay:Nnn #1 #2 #3
  {
    \__hook_debug:n { \iow_term:n { ->~Add~generic~cmd~hook~for~#2~(#3). } }
    \__hook_debug:n
      { \iow_term:n { !~In~the~preamble:~delaying. } }
    \prop_gput:Nnn \g__hook_delayed_patches_prop { #2 / #3 }
      { \__hook_cmd_try_patch:nn {#2} {#3} }
  }
```

The delayed patches are added to a property list to prevent duplication, and the code stored in the property list for each key is executed. The function `\__hook_patch_cmd_or_delay:Nnn` is also redefined to be `\__hook_patch_command:Nnn` so that no further delaying is attempted.

```
\cs_new_protected:Npn \__hook_cmd_begindocument_code:
  {
    \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:
  }
\g@addto@macro \@kernel@after@begindocument
  { \__hook_cmd_begindocument_code: }
```

(*End definition for* `\__hook_patch_cmd_or_delay:Nnn` *and* `\__hook_cmd_begindocument_code:`.)

`\__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_new_protected:Npn \__hook_cmd_try_patch:nn #1 #2
  {
    \__hook_debug:n
      { \iow_term:x { ->~\string\begin{document}~try~cmd / #1 / #2. } }
    \__hook_if_declared:nTF { cmd / #1 / #2 }
      {
        \__hook_debug:n
          { \iow_term:n { .->~Giving~up:~hook~already~created. } }
      }
      {
        \cs_if_exist:cT {#1}
          { \exp_args:Nc \__hook_patch_command:Nnn {#1} {#1} {#2} }
      }
  }
```

(*End definition for* `\__hook_cmd_try_patch:nn`.)

4.5 Patching commands

`\__hook_patch_command:Nnn`
`\__hook_patch_check:NNnn`
`\__hook_if_public_command:NTF`
`\__hook_if_public_command:w`

`\__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_debug:x { \token_to_str:N #1 = \token_to_meaning: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 #1 #2 #3 #4
  {
    #1 #2 {#4}
      {
        \__kernel_msg_error:nnxx { hooks } { cant-patch }
          { \token_to_str:N #2 } {#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.

```
\use:x
  {
    \prg_new_protected_conditional:Npnn
        \exp_not:N \__hook_if_public_command:N ##1 { TF }
      {
        \exp_not:N \exp_last_unbraced:Nf
          \exp_not:N \__hook_if_public_command:w
            { \exp_not:N \cs_to_str:N ##1 }
          \tl_to_str:n { _ _ } \s__hook_mark
      }
  }
\exp_last_unbraced:NNNNo
\cs_new_protected:Npn \__hook_if_public_command:w
    #1 \tl_to_str:n { _ _ } #2 \s__hook_mark
  {
    \tl_if_empty:nTF {#2}
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\__hook_patch_command:Nnn` *and others.*)

4.5.1 Patching by expansion and redefinition

`\g_hook_patch_action_list_tl`

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
  {
    { \@if@DeclareRobustCommand \__hook_patch_DeclareRobustCommand:Nnn }
    { \@if@newcommand \__hook_patch_newcommand:Nnn }
    { \__kernel_cmd_if_xparse:NTF \__hook_cmd_patch_xparse:Nnn }
  }
```

(*End definition for* `\g_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`). If so,

we forward the action to `\__hook_patch_newcommand:Nnn`, otherwise call the patching engine `\__hook_patch_expand_redefine:NNnn` with a `\c_false_bool` to indicate that there is no optional argument.

```
\cs_new_protected:Npn \__hook_patch_DeclareRobustCommand:Nnn #1
  {
    \exp_args:Nc \@if@newcommand { \cs_to_str:N #1 ~ }
      { \exp_args:Nc \__hook_patch_newcommand:Nnn }
      { \exp_args:NNc \__hook_patch_expand_redefine:NNnn \c_false_bool }
        { \cs_to_str:N #1 ~ }
  }
```

(*End definition for* `\__hook_patch_DeclareRobustCommand:Nnn`.)

`\__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
      { \c_backslash_str \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`.)

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

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

We'll start by counting the number of arguments in the command by counting the number of characters in the `\cs_argument_spec:N` of the macro, divided by two, and subtracting one if the command has an optional argument (that is, an extra `[]` in its ⟨*parameter text*⟩).

```
    \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 }
      }
```

Now build two token lists:

`\l__hook_patch_param_text_tl` will contain the ⟨*parameter text*⟩ to be used when redefining the macro. It should be identical to the ⟨*parameter text*⟩ used when originally defining that macro.

`\l__hook_patch_replacement_tl` will contain braced pairs of $\texttt{\#}_{12}\langle num\rangle$ to feed to the macro when expanded. This token list as well as the previous will have the first item surrounded by `[...]` in the case of an optional argument.

```
\int_compare:nNnTF { \l__hook_patch_num_args_int } > { \c_zero_int }
  {
    \tl_set:Nx \l__hook_patch_param_text_tl
      { \bool_if:NTF #1 { [####1] } {  ####1  } }
    \tl_set:Nx \l__hook_patch_replacement_tl
      { \bool_if:NTF #1 { [####1] } { {####1} } }
    \int_step_inline:nnn { 2 } { \l__hook_patch_num_args_int }
      {
        \tl_put_right:Nn \l__hook_patch_param_text_tl    { ## ####1 }
        \tl_put_right:Nn \l__hook_patch_replacement_tl { { ## ####1 } }
      }
  }
  {
    \tl_clear:N \l__hook_patch_param_text_tl
    \tl_clear:N \l__hook_patch_replacement_tl
  }
```

Finally, before redefining, we need to also get the prefixes used when defining the command. Here we ensure that the `\escapechar` is printable, otherwise a macro defined with prefixes `\protected \long` will have it `\meaning` printed as `protectedlong`, making life unnecessarily complicated. Here the `\escapechar` is changed to `/`, then we loop between pairs of `/.../` extracting the prefixes.

```
\group_begin:
  \int_set:Nn \tex_escapechar:D { `\/ }
  \use:x
    {
\group_end:
\tl_set:Nx \exp_not:N \l__hook_patch_prefixes_tl
  { \exp_not:N \__hook_make_prefixes:w \cs_prefix_spec:N #2 / / }
    }
```

Now that all the needed tools are ready, without further ado we'll redefine the command `#2`. The definition uses the prefixes gathered in `\l__hook_patch_prefixes_tl`, a primitive `\tex_def:D` to avoid adding extra prefixes, and the $\langle$*parameter text*$\rangle$ from `\l__hook_patch_param_text_tl`.

Then finally, in the body of the definition, we insert `cmd/#3/before` if `#4` is `before`, the code of the command expanded once grabbing the parameters in `\l__hook_patch_replacement_tl`, and `cmd/#3/after` if `#4` is `after`.

```
\use:x
  {
    \l__hook_patch_prefixes_tl \tex_def:D
        \exp_not:N #2 \exp_not:V \l__hook_patch_param_text_tl
      {
        \str_if_eq:nnT {#4} { before }
          { \exp_not:N \UseHook { cmd / #3 / #4 } }
        \exp_args:No \exp_not:o
          { \exp_after:wN #2 \l__hook_patch_replacement_tl }
        \str_if_eq:nnT {#4} { after }
          { \exp_not:N \UseHook { cmd / #3 / #4 } }
      }
```

```
    }
  }
```

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 /
      }
  }
```

(*End definition for* `\__hook_patch_expand_redefine:NNnn` *and* `\__hook_make_prefixes:w`.)

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 `\meaning` of the command, doing the necessary manipulations on the generated string, and the retokenizing that again by using `\scantokens`.

Patching by retokenization is definitely a riskier business, because it relies that the tokens printed by `\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.

`\__hook_retokenize_patch:Nnn`

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 `\robust@command@act` and the top-level macros would be incorrectly patched. In that case, we just check if the `\cs_argument_spec:N` is empty, and call `\__hook_patch_expand_redefine:NNnn`.

```
\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 `\__hook_try_patch_with_catcodes:Nnnnw` 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
        {
          \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 } % _:
            {
              \exp_not:V \l__hook_tmpa_tl   % @
              \exp_not:V \l__hook_tmpb_tl   % _:
            }
        }
            \q_recursion_tail \q_recursion_stop
```

If no catcode setting succeeds, give up and raise an error. The command isn't changed in any way in that case.

```
        {
          \__kernel_msg_error:nnxx { hooks } { cant-patch }
            { \c_backslash_str #2 } { retok }
        }
    }
}
```

(*End definition for* `\__hook_retokenize_patch:Nnn`.)

`\__hook_try_patch_with_catcodes:Nnnnw` 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:Nnnnw #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 }
        \__hook_patch_retokenize:Nnnn #1 {#2} {#3} {#4}
        \use_i_delimit_by_q_recursion_stop:nw \use_none:n
      }
      {
        \__hook_patch_debug:x { --~macro~cannot~be~retokenized~cleanly }
        \__hook_try_patch_with_catcodes:Nnnnw #1 {#2} {#3}
      }
  }
```

(*End definition for* `\__hook_try_patch_with_catcodes:Nnnnw`.)

`\kerneltmpDoNotUse` This is an oddity required to be safe (as safe as reasonably possible) when patching the command. The entirety of

⟨*prefixes*⟩ `\def` ⟨*cs*⟩ ⟨*parameter text*⟩ `{`⟨*replacement text*⟩`}`

will go through `\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 `\def` or the ⟨*prefixes*⟩. That is why the definitions that will eventually go into `\scantokens` will use the oddly (but hopefully clearly)-named `\kerneltmpDoNotUse`:

```
\cs_new_eq:NN \kerneltmpDoNotUse !
```

PhO: Maybe this can be avoided by running the ⟨*parameter text*⟩ *and the* ⟨*replacement text*⟩ *separately through* `\scantokens` *and then putting everything together at the end.*

(*End definition for* `\kerneltmpDoNotUse`.)

`\__hook_patch_required_catcodes:` Here are the catcode settings that are *mandatory* when retokenizing commands. These are the minimum necessary settings to perform the definitions: they identify control sequences, which must be escaped with $\backslash_0$, delimit the definition with $\{_1$ and $\}_2$, and mark parameters with $\#_6$. Everything else may be changed, but not these.

```
\cs_new_protected:Npn \__hook_patch_required_catcodes:
  {
    \char_set_catcode_escape:N \\
    \char_set_catcode_group_begin:N \{
    \char_set_catcode_group_end:N \}
    \char_set_catcode_parameter:N \#
    % \int_set:Nn \tex_endlinechar:D { -1 }
    % \int_set:Nn \tex_newlinechar:D { -1 }
  }
```

PhO: etoolbox sets the `\endlinechar` *and* `\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* `\newlinechar=-1` *avoids a space token being added after the definition.*

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

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

(*End definition for* `\__hook_patch_required_catcodes:`.)

`\__hook_cmd_if_scanable:NnTF` 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.

```
\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 definition for* `\__hook_cmd_if_scanable:NnTF`.)

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

```
\cs_new_protected:Npn \__hook_patch_retokenize:Nnnn #1 #2 #3 #4
  {
```

Start off by making some things `\relax` to avoid lots of `\noexpand` below.

```
    \cs_set_eq:NN \kerneltmpDoNotUse \scan_stop:
    \cs_set_eq:NN \__hook_tmp:w \scan_stop:
    \use:x
      {
```

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 `before` or `after` hooks around the ⟨*replacement text*⟩: accordingly.

```
        \cs_set:Npn \__hook_tmp:w
            ####1 \tl_to_str:n { macro: } ####2 -> ####3 \s__hook_mark
          {
            ####1 \def \kerneltmpDoNotUse ####2
              {
                \str_if_eq:nnT {#3} { before }
                  { \token_to_str:N \UseHook { cmd / #2 / #3 } }
                ####3
                \str_if_eq:nnT {#3} { after }
                  { \token_to_str:N \UseHook { cmd / #2 / #3 } }
              }
          }
```

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

```
        \tl_set:Nx \exp_not:N \l__hook_tmpa_tl
          { \exp_not:N \__hook_tmp:w \token_to_meaning:N #1 \s__hook_mark }
      }
```

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.

```
    \tl_rescan:nV { #4 \__hook_patch_required_catcodes: } \l__hook_tmpa_tl
```

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

```
    \cs_set_eq:NN #1 \kerneltmpDoNotUse
  }
```

(*End definition for* `\__hook_patch_retokenize:Nnnn`.)

4.6 Messages

```
\__kernel_msg_new:nnnn { hooks } { wrong-cmd-hook }
  {
    Command~hook~‘cmd/#1/#2’~invalid.\\
    The~hook~should~be~‘cmd/#1/before’~or~‘cmd/#1/after’.
  }
  {
    You~tried~to~add~a~hook~to~command~\iow_char:N \\#1,~but~‘#2’~
    is~an~invalid~position.~Only~‘before’~or~‘after’~are~allowed.
  }
\__kernel_msg_new:nnnn { hooks } { cant-patch }
  {
    Command~‘#1’~cannot~have~hooks~because~it~
    \__hook_unpatchable_cases:n {#2} .
  }
  {
    You~tried~to~add~a~hook~to~‘#1’,~but~LaTeX~was~not~able~to~
    add~the~hook~to~that~command~because~‘#1’~
    \__hook_unpatchable_cases:n {#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 }
      }
  }
⟨latexrelease⟩\IncludeInRelease{0000/00/00}{ltcmdhooks}%
⟨latexrelease⟩                  {The~hook~management~system~for~commands}
⟨latexrelease⟩
```

The command `\__hook_cmd_begindocument_code:` is used in an internal hook, so we need to make sure it has a harmless definition after rollback as that will not remove it from the kernel hook.

```
⟨latexrelease⟩\cs_set_eq:NN \__hook_cmd_begindocument_code: \prg_do_nothing:
⟨latexrelease⟩
⟨latexrelease⟩\EndModuleRelease
\ExplSyntaxOff
⟨/2ekernel | latexrelease⟩
⟨@@=⟩
```

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.