The `lthooks` package*

Frank Mittelbach†

July 21, 2020

Contents

*This package has version v0.9b dated 2020/07/19, © LaTeX Project.
†Code improvements for speed and other goodies by Phelype Oleinik

1 Introduction

Hooks are points in the code of commands or environments where it is possible to add processing code into existing commands. This can be done by different packages that do not know about each other and to allow for hopefully safe processing it is necessary to sort different chunks of code added by different packages into a suitable processing order.

This is done by the packages adding chunks of code (via `\AddToHook`) and labeling their code with some label by default using the package name as a label.

At `\begin{document}` all code for a hook is then sorted according to some rules (given by `\DeclareHookRule`) for fast execution without processing overhead. If the hook code is modified afterwards (or the rules are changed), a new version for fast processing is generated.

Some hooks are used already in the preamble of the document. If that happens then the hook is prepared for execution (and sorted) already at that point.

2 Package writer interface

The hook management system is offered as a set of CamelCase commands for traditional LaTeX 2ε packages (and for use in the document preamble if needed) as well as `expl3` commands for modern packages, that use the L3 programming layer of LaTeX. Behind the scenes, a single set of data structures is accessed so that packages from both worlds can coexist and access hooks in other packages.

2.1 LaTeX 2_ε interfaces

2.1.1 Declaring hooks and using them in code

With two exceptions, hooks have to be declared before they can be used. The exceptions are hooks in environments (i.e., executed at `\begin` and `\end`) and hooks run when loading files, e.g. before and after a package is loaded, etc. Their hook names depend on the environment or the file name and so declaring them beforehand is difficult.

`\NewHook` `\NewHook {⟨hook⟩}`

Creates a new ⟨*hook*⟩. If this is a hook provided as part of a package it is suggested that the ⟨*hook*⟩ name is always structured as follows: ⟨*package-name*⟩`/`⟨*hook-name*⟩. If necessary you can further subdivide the name by adding more `/` parts. If a hook name is already taken, an error is raised and the hook is not created.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\NewReversedHook` `\NewReversedHook {⟨hook⟩}`

Like `\NewHook` declares a new ⟨*hook*⟩. the difference is that the code chunks for this hook are in reverse order by default (those added last are executed first). Any rules for the hook are applied after the default ordering. See sections 2.3 and 2.4 for further details.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\NewMirroredHookPair` `\NewMirroredHookPair {⟨hook-1⟩} {⟨hook-2⟩}`

A shorthand for `\NewHook{`⟨*hook-1*⟩`}\NewReversedHook{`⟨*hook-2*⟩`}`.

The ⟨*hooks*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\UseHook` `\UseHook {⟨hook⟩}`

Execute the hook code inside a command or environment.[1]

Before `\begin{document}` the fast execution code for a hook is not set up, so in order to use a hook there it is explicitly initialized first. As that involves assignments using a hook at those times is not 100% the same as using it after `\begin{document}`.

The ⟨*hook*⟩ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

`\UseOneTimeHook` `\UseOneTimeHook {⟨hook⟩}`

Some hooks are only used (and can be only used) in one place, for example, those in `\begin{document}` or `\end{document}`. Once we have passed that point adding to the hook through a defined `\`⟨*addto-cmd*⟩ command (e.g., `\AddToHook` or `\AtBeginDocument`, etc.) would have no effect (as would the use of such a command inside the hook code itself). It is therefore customary to redefine `\`⟨*addto-cmd*⟩ to simply process its argument, i.e., essentially make it behave like `\@firstofone`.

`\UseOneTimeHook` does that: it records that the hook has been consumed and any further attempt to add to it will result in executing the code to be added immediately.

FMi: Maybe add an error version as well?

The ⟨*hook*⟩ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

2.1.2 Updating code for hooks

`\AddToHook` `\AddToHook {⟨hook⟩}[⟨label⟩]{⟨code⟩}`

Adds ⟨*code*⟩ to the ⟨*hook*⟩ labeled by ⟨*label*⟩. If the optional argument ⟨*label*⟩ is not provided, if `\AddToHook` is used in a package/class, then the current package/class name is used, otherwise `top-level` is used (see section 2.1.3).

If there already exists code under the ⟨*label*⟩ then the new ⟨*code*⟩ is appended to the existing one (even if this is a reversed hook). If you want to replace existing code under the ⟨*label*⟩, first apply `\RemoveFromHook`.

The hook doesn't have to exist for code to be added to it. However, if it is not declared later then obviously the added ⟨*code*⟩ will never be executed. This allows for hooks to work regardless of package loading order and enables packages to add to hook of other packages without worrying whether they are actually used in the current document. See section 2.1.5.

The ⟨*hook*⟩ and ⟨*label*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\RemoveFromHook` `\RemoveFromHook {⟨hook⟩}[⟨label⟩]`

Removes any code labeled by ⟨*label*⟩ from the ⟨*hook*⟩. If the optional argument ⟨*label*⟩ is not provided, if `\AddToHook` is used in a package/class, then the current package/class name is used, otherwise `top-level` is used.

If the optional argument is `*`, then all code chunks are removed. This is rather dangerous as it drops code from other packages one may not know about!

The ⟨*hook*⟩ and ⟨*label*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

A useful application for this declaration inside the document body is when one wants to temporarily add code to hooks and later remove it again, e.g.,

```
\AddToHook{env/quote/before}{\small}
\begin{quote}
  A quote set in a smaller typeface
\end{quote}
...
\RemoveFromHook{env/quote/before}
... now back to normal for further quotes
```

Note that you can't cancel the setting with

```
\AddToHook{env/quote/before}{}
```

because that only "adds" a further empty chunk of code to the hook. Adding `\normalsize` would work but that means the hook then contained `\small\normalsize` which means to font size changes for no good reason.

The above is only needed if one wants to typeset several quotes in a smaller typeface. If the hook is only needed once then `\AddToHookNext` is simpler, because it resets itself after one use.

\AddToHookNext `\AddToHookNext {⟨hook⟩}{⟨code⟩}`

Adds ⟨*code*⟩ to the next invocation of the ⟨*hook*⟩. The code is executed after the normal hook code has finished and it is executed only once, i.e. it is deleted after it was used.

Using the declaration is a global operation, i.e., the code is not lost, even if the declaration is used inside a group and the next invocation happens after the group. If the declaration is used several times before the hook is executed then all code is executed in the order in which it was declared.[2]

The hook doesn't have to exist for code to be added to it. This allows for hooks to work regardless of package loading order. See section 2.1.5.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

2.1.3 Hook names and default labels

It is best practice to use `\AddToHook` in packages or classes *without specifying a* ⟨*label*⟩ because then the package or class name is automatically used, which is helpful if rules are needed, and avoids mistyping the ⟨*label*⟩.

Using an explicit ⟨*label*⟩ is only necessary in very specific situations, e.g., if you want to add several chunks of code into a single hook and have them placed in different parts of the hook (by providing some rules).

The other case is when you develop a larger package with several sub-packages. In that case you may want to use the same ⟨*label*⟩ throughout the sub-packages in order to avoid that the labels change if you internally reorganize your code.

It is not enforced, but highly recommended that the hooks defined by a package, and the ⟨*labels*⟩ used to add code to other hooks contain the package name to easily identify the source of the code chunk and to prevent clashes. This should be the standard practice, so this hook management code provides a shortcut to refer to the current package in the name of a ⟨*hook*⟩ and in a ⟨*label*⟩. If ⟨*hook*⟩ name or ⟨*label*⟩ consist just of a single dot (`.`), or starts with a dot followed by a slash (`./`) then the dot denotes the ⟨*default label*⟩ (usually the current package or class name—see `\DeclareDefaultHookLabel`). A "`.`" or "`./`" anywhere else in a ⟨*hook*⟩ or in ⟨*label*⟩ is treated literally and is not replaced.

For example, inside the package `mypackage.sty`, the default label is `mypackage`, so the instructions:

```
\NewHook   {./hook}
\AddToHook {./hook}[.]{code}     % Same as \AddToHook{./hook}{code}
\AddToHook {./hook}[./sub]{code}
\DeclareHookRule{begindocument}{.}{<}{babel}
\AddToHook {file/after/foo.tex}{code}
```

are equivalent to:

```
\NewHook   {mypackage/hook}[mypackage]{code}
\AddToHook {mypackage/hook}[mypackage]{code}
\AddToHook {mypackage/hook}[mypackage/sub]{code}
\DeclareHookRule{begindocument}{mypackage}{<}{babel}
\AddToHook {file/after/foo.tex}{code}                    % unchanged
```

[2] There is no mechanism to reorder such code chunks (or delete them).

The ⟨*default label*⟩ is automatically set to the name of the current package or class (using `\@currname`). If `\@currname` is not set (because the hook command is used outside of a package, or the current file wasn't loaded with `\usepackage` or `\documentclass`), then the `top-level` is used as the ⟨*default label*⟩.

This syntax is available in all ⟨*label*⟩ arguments and most ⟨*hook*⟩, both in the LaTeX 2_ε interface, and the LaTeX3 interface described in section 2.2.

Note, however, that the replacement of `.` by the ⟨*default label*⟩ takes place when the hook command is executed, so actions that are somehow executed after the package ends will have the wrong ⟨*default label*⟩ if the dot-syntax is used. For that reason, this syntax is not available in `\UseHook` (and `\hook_use:n`) because the hook is most of the time used outside of the package file in which it was defined. This syntax is also not available in the hook conditionals `\IfHookEmptyTF` (and `\hook_if_empty:nTF`) and `\IfHookExistTF` (and `\hook_if_exist:nTF`) because these conditionals are used in some performance-critical parts of the hook management code, and because they are usually used to refer to other package's hooks, so the dot-syntax doesn't make much sense.

In some cases, for example in large packages, one may want to separate it in logical parts, but still use the main package name as ⟨*label*⟩, then the ⟨*default label*⟩ can be set using `\DeclareDefaultHookLabel`:

`\DeclareDefaultHookLabel` `\DeclareDefaultHookLabel {`⟨*default label*⟩`}`

Sets the ⟨*default label*⟩ to be used in ⟨*label*⟩ arguments. If `\DeclareDefaultHookLabel` is not used in the current package, `\@currname` is used instead. If `\@currname` is not set, the code is assumed to be in the main document, in which case `top-level` is used.

The effect of `\DeclareDefaultHookLabel` holds for the current file, and is reset to the previous value when the file is closed.

2.1.4 Defining relations between hook code

The default assumption is that code added to hooks by different packages is independent and the order in which it is executed is irrelevant. While this is true in many case it is obviously false in many others.

Before the hook management system was introduced packages had to take elaborate precaution to determine of some other package got loaded as well (before or after) and find some ways to alter its behavior accordingly. In addition is was often the user's responsibility to load packages in the right order so that code added to hooks got added in the right order and some cases even altering the loading order wouldn't resolve the conflicts.

With the new hook management system it is now possible to define rules (i.e., relationships) between code chunks added by different packages and explicitly describe in which order they should be processed.

`\DeclareHookRule` `\DeclareHookRule {⟨hook⟩}{⟨label1⟩}{⟨relation⟩}{⟨label2⟩}`

Defines a relation between ⟨*label1*⟩ and ⟨*label2*⟩ for a given ⟨*hook*⟩. If ⟨*hook*⟩ is `??` this defines a relation for all hooks that use the two labels, i.e., that have chunks of code labeled with ⟨*label1*⟩ and ⟨*label2*⟩. Rules specific to a given hook take precedence over default rules that use `??` as the ⟨*hook*⟩.

Currently, the supported relations are the following:

`before` or `<` Code for ⟨*label1*⟩ comes before code for ⟨*label2*⟩.

`after` or `>` Code for ⟨*label1*⟩ comes after code for ⟨*label2*⟩.

`incompatible-warning` Only code for either ⟨*label1*⟩ or ⟨*label2*⟩ can appear for that hook (a way to say that two packages—or parts of them—are incompatible). A warning is raised if both labels appear in the same hook.

`incompatible-error` Like `incompatible-error` but instead of a warning a LaTeX error is raised, and the code for both labels are dropped from that hook until the conflict is resolved.

`removes` Code for ⟨*label1*⟩ overwrites code for ⟨*label2*⟩. More precisely, code for ⟨*label2*⟩ is dropped for that hook. This can be used, for example if one package is a superset in functionality of another one and therefore wants to undo code in some hook and replace it with its own version.

`unrelated` The order of code for ⟨*label1*⟩ and ⟨*label2*⟩ is irrelevant. This rule is there to undo an incorrect rule specified earlier.

The ⟨*hook*⟩ and ⟨*label*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\ClearHookRule` `\ClearHookRule{⟨hook⟩}{⟨label1⟩}{⟨label2⟩}`

Syntactic sugar for saying that ⟨*label1*⟩ and ⟨*label2*⟩ are unrelated for the given ⟨*hook*⟩.

`\DeclareDefaultHookRule` `\DeclareDefaultHookRule{⟨label1⟩}{⟨relation⟩}{⟨label2⟩}`

This sets up a relation between ⟨*label1*⟩ and ⟨*label2*⟩ for all hooks unless overwritten by a specific rule for a hook. Useful for cases where one package has a specific relation to some other package, e.g., is `incompatible` or always needs a special ordering `before` or `after`. (Technically it is just a shorthand for using `\DeclareHookRule` with `??` as the hook name.)

Declaring default rules is only supported in the document preamble.[3]

The ⟨*label*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

2.1.5 Querying hooks

Simpler data types, like token lists, have three possible states; they can:

- exist and be empty;
- exist and be non-empty; and

[3] Trying to do so, e.g., via `\DeclareHookRule` with `??` has bad side-effects and is not supported (though not explicitly caught for performance reasons).

- not exist (in which case emptiness doesn't apply);

Hooks are a bit more complicated: they have four possible states. A hook may exist or not, and either way it may or may not be empty. This means that even a hook that doesn't exist may be non-empty.

This seemingly strange state may happen when, for example, package *A* defines hook `A/foo`, and package *B* adds some code to that hook. However, a document may load package *B* before package *A*, or may not load package *A* at all. In both cases some code is added to hook `A/foo` without that hook being defined yet, thus that hook is said to be non-empty, whereas it doesn't exist. Therefore, querying the existence of a hook doesn't imply its emptiness, neither does the other way around.

A hook is said to be empty when no code was added to it, either to its permanent code pool, or to its "next" token list. The hook doesn't need to be declared to have code added to its code pool. A hook is said to exist when it was declared with `\NewHook` or some variant thereof.

`\IfHookEmptyTF` ⋆ `\IfHookEmptyTF {⟨hook⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the ⟨*hook*⟩ is empty (*i.e.*, no code was added to it using either `\AddToHook` or `\AddToHookNext`), and branches to either ⟨*true code*⟩ or ⟨*false code*⟩ depending on the result.

The ⟨*hook*⟩ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

`\IfHookExistTF` ⋆ `\IfHookExistTF {⟨hook⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the ⟨*hook*⟩ exists (if it was created with either `\NewHook`, `\NewReversedHook`, or `\NewMirroredHookPair`), and branches to either ⟨*true code*⟩ or ⟨*false code*⟩ depending on the result.

The existence of a hook usually doesn't mean much from the viewpoint of code that tries to add/remove code from that hook, since package loading order may vary, thus the creation of hooks is asynchronous to adding and removing code from it, so this test should be used sparingly.

The ⟨*hook*⟩ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

FMi: Would be helpful if we provide some use cases

2.1.6 Displaying hook code

If one has to adjust the code execution in a hook using a hook rule it is helpful to get some information about the code associated with a hook, its current order and the existing rules.

\ShowHook `\ShowHook {⟨hook⟩}`

Displays information about the ⟨*hook*⟩ such as

- the code chunks (and their labels) added to it,
- any rules set up to order them,

  FMi: currently this is missing the default rules that apply, guess that needs fixing

- the computed order (if already defined),
- any code executed on the next invocation only.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

2.1.7 Debugging hook code

\DebugHookOn `\DebugHookOn`
\DebugHookOff

Turn the debugging of hook code on or off. This displays changes made to the hook data structures. The output is rather coarse and not really intended for normal use.

2.2 L3 programming layer (expl3) interfaces

This is a quick summary of the LaTeX3 programming interfaces for use with packages written in `expl3`. In contrast to the LaTeX 2ε interfaces they always use mandatory arguments only, e.g., you always have to specify the ⟨*label*⟩ for a code chunk. We therefore suggest to use the declarations discussed in the previous section even in `expl3` packages, but the choice is yours.

\hook_new:n `\hook_new:n{⟨hook⟩}`
\hook_new_reversed:n `\hook_new_pair:nn{⟨hook-1⟩}{⟨hook-2⟩}`
\hook_new_pair:nn

Creates a new ⟨*hook*⟩ with normal or reverse ordering of code chunks. `\hook_new_pair:nn` creates a pair of such hooks with `{⟨hook-2⟩}` being a reversed hook. If a hook name is already taken, an error is raised and the hook is not created.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

\hook_use:n `\hook_use:n {⟨hook⟩}`

Executes the `{⟨hook⟩}` code followed (if set up) by the code for next invocation only, then empties that next invocation code.

The ⟨*hook*⟩ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

\hook_use_once:n `\hook_use_once:n {⟨hook⟩}`

Changes the `{⟨hook⟩}` status so that from now on any addition to the hook code is executed immediately. Then execute any `{⟨hook⟩}` code already set up.

FMi: better L3 name?

The ⟨*hook*⟩ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

`\hook_gput_code:nnn` `\hook_gput_code:nnn {⟨hook⟩} {⟨label⟩} {⟨code⟩}`

Adds a chunk of ⟨*code*⟩ to the ⟨*hook*⟩ labeled ⟨*label*⟩. If the label already exists the ⟨*code*⟩ is appended to the already existing code.

If code is added to an external ⟨*hook*⟩ (of the kernel or another package) then the convention is to use the package name as the ⟨*label*⟩ not some internal module name or some other arbitrary string.

The ⟨*hook*⟩ and ⟨*label*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\hook_gput_next_code:nn` `\hook_gput_next_code:nn {⟨hook⟩} {⟨code⟩}`

Adds a chunk of ⟨*code*⟩ for use only in the next invocation of the ⟨*hook*⟩. Once used it is gone.

This is simpler than `\hook_gput_code:nnn`, the code is simply appended to the hook in the order of declaration at the very end, i.e., after all standard code for the hook got executed.

Thus if one needs to undo what the standard does one has to do that as part of ⟨*code*⟩.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\hook_gremove_code:nn` `\hook_gremove_code:nn {⟨hook⟩} {⟨label⟩}`

Removes any code for ⟨*hook*⟩ labeled ⟨*label*⟩.

If the code for that ⟨*label*⟩ wasn't yet added to the ⟨*hook*⟩, an order is set so that when some code attempts to add that label, the removal order takes action and the code is not added.

If the second argument is `*`, then all code chunks are removed. This is rather dangerous as it drops code from other packages one may not know about, so think twice before using that!

The ⟨*hook*⟩ and ⟨*label*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\hook_gset_rule:nnnn` `\hook_gset_rule:nnnn {⟨hook⟩} {⟨label1⟩} {⟨relation⟩} {⟨label2⟩}`

Relate ⟨*label1*⟩ with ⟨*label2*⟩ when used in ⟨*hook*⟩. See `\DeclareHookRule` for the allowed ⟨*relation*⟩s. If ⟨*hook*⟩ is `??` a default rule is specified.

The ⟨*hook*⟩ and ⟨*label*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.3. The dot-syntax is parsed in both ⟨*label*⟩ arguments, but it usually makes sense to be used in only one of them.

`\hook_if_empty_p:n` ⋆
`\hook_if_empty:nTF` ⋆
`\hook_if_empty:n {⟨hook⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the ⟨*hook*⟩ is empty (*i.e.*, no code was added to it using either `\AddToHook` or `\AddToHookNext`), and branches to either ⟨*true code*⟩ or ⟨*false code*⟩ depending on the result.

The ⟨*hook*⟩ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

`\hook_if_exist_p:n` ⋆
`\hook_if_exist:nTF` ⋆

`\hook_if_exist:n {`⟨*hook*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Tests if the ⟨*hook*⟩ exists (if it was created with either `\NewHook`, `\NewReversedHook`, or `\NewMirroredHookPair`), and branches to either ⟨*true code*⟩ or ⟨*false code*⟩ depending on the result.

FMi: what are the results for generic hooks that do not need to be declared?

The existence of a hook usually doesn't mean much from the viewpoint of code that tries to add/remove code from that hook, since package loading order may vary, thus the creation of hooks is asynchronous to adding and removing code from it, so this test should be used sparingly.

The ⟨*hook*⟩ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

`\hook_debug_on:`
`\hook_debug_off:`

`\hook_debug_on:`

Turns the debugging of hook code on or off. This displays changes to the hook data.

2.3 On the order of hook code execution

Chunks of code for a ⟨*hook*⟩ under different labels are supposed to be independent if there are no special rules set up that define a relation between the chunks. This means that you can't make assumptions about the order of execution!

Suppose you have the following declarations:

```
\NewHook{myhook}
\AddToHook{myhook}[packageA]{\typeout{A}}
\AddToHook{myhook}[packageB]{\typeout{B}}
\AddToHook{myhook}[packageC]{\typeout{C}}
```

then executing the hook with `\UseHook` will produce the typeout `A B C` in that order. In other words, the execution order is computed to be `packageA`, `packageB`, `packageC` which you can verify with `\ShowHook{myhook}`:

```
The hook 'myhook':
 Code chunks:
    packageA -> \typeout {A}
    packageB -> \typeout {B}
    packageC -> \typeout {C}
 Extra code next invocation:
    ---
 Rules:
    ---
 Execution order:
    packageA, packageB, packageC
```

The reason is that the code chunks are internally saved in a property list and the initial order of such a property list is the order in which key-value pairs got added. However, that is only true if nothing other than adding happens!

Suppose, or example, you want to replace the code chunk for `packageA`, e.g.,

```
\RemoveFromHook{myhook}[packageA]
\AddToHook{myhook}[packageA]{\typeout{A alt}}
```

then your order becomes `packageB`, `packageC`, `packageA` because the label got removed from the property list and then re-added (at its end).

While that may not be too surprising, the execution order is also sometimes altered if you add a redundant rule, e.g. if you specify

```
\DeclareHookRule{myhook}{packageA}{before}{packageB}
```

instead of the previous lines we get

```
The hook 'myhook':
 Code chunks:
    packageA -> \typeout {A}
    packageB -> \typeout {B}
    packageC -> \typeout {C}
 Extra code next invocation:
    ---
 Rules:
    packageA|packageB with relation before
 Execution order (after applying rules):
    packageA, packageC, packageB
```

As you can see the code chunks are still in the same order, but in the execution order for the labels `packageB` and `packageC` have swapped places. The reason is that, with the rule there are two orders that satisfy it, and the algorithm for sorting happened to pick a different one compared to the case without rules (where it doesn't run at all as there is nothing to resolve). Incidentally, if we had instead specified the redundant rule

```
\DeclareHookRule{myhook}{packageB}{before}{label-3}
```

the execution order would not have changed.

In summary: it is not possible to rely on the order of execution unless there are rules that partially or fully define the order (in which you can rely on them being fulfilled).

2.4 The use of "reversed" hooks

You may have wondered why you can declare a "reversed" hook with `\NewReversedHook` and what that does exactly.

In short: the execution order of a reversed hook (without any rules!) is exactly reversed to the order you would have gotten for a hook declared with `\NewHook`.

This is helpful if you have a pair of hooks where you expect to see code added that involves grouping, e.g., starting an environment in the first and closing that environment in the second hook. To give a somewhat contrived example[4], suppose there is a package adding the following:

```
\AddToHook{env/quote/before}[package-1]{\begin{itshape}}
\AddToHook{env/quote/after} [package-1]{\end{itshape}}
```

As a result, all quotes will be in italics. Now suppose further that the user wants the quotes also in blue and therefore adds:

[4] there are simpler ways to achieve the same effect.

```
\usepackage{color}
\AddToHook{env/quote/before}{\begin{color}{blue}}
\AddToHook{env/quote/after} {\end{color}}
```

Now if the `env/quote/after` hook would be a normal hook we would get the same execution order in both hooks, namely:

```
package-1, top-level
```

(or vice versa) and as a result, would get:

```
\begin{itshape}\begin{color}{blue} ...
\end{itshape}\end{color}
```

and an error message that `\begin{color}` ended by `\end{itshape}`. With `env/quote/after` declared as a reversed hook the execution order is reversed and so all environments are closed in the correct sequence and `\ShowHook` would give us the following output:

```
The hook 'env/quote/after':
 Code chunks:
    package-1 -> \end {itshape}
    top-level -> \end {color}
 Extra code next invocation:
    ---
 Rules:
    ---
 Execution order (after reversal):
    top-level, package-1
```

The reversal of the execution order happens before applying any rules, so if you alter the order you will probably have to alter it in both hooks, not just in one, but that depends on the use case.

2.5 Private LaTeX kernel hooks

There are a few places where it is absolutely essential for LaTeX to function correctly that code is executed in a precisely defined order. Even that could have been implemented with the hook management (by adding various rules to ensure the appropriate ordering with respect to other code added by packages). However, this makes every document unnecessary slow, because there has to be sorting even through the result is predetermined. Furthermore it forces package writers to unnecessarily add such rules if they add further code to the hook (or break LaTeX).

For that reason such code is not using the hook management, but instead private kernel commands directly before or after a public hook with the following naming convention: `\@kernel@before@`⟨*hookname*⟩ or `\@kernel@after@`⟨*hookname*⟩. For example, in `\enddocument` you find

```
\UseHook{enddocument}%
\@kernel@after@enddocument
```

which means first the user/package-accessible `enddocument` hook is executed and then the internal kernel hook. As their name indicates these kernel commands should not be altered by third-party packages, so please refrain from that in the interest of stability and instead use the public hook next to it.[5]

[5] As with everything in TeX there is not enforcement of this rule, and by looking at the code it is

2.6 Legacy LaTeX 2ε interfaces

LaTeX 2ε offered a small number of hooks together with commands to add to them. They are listed here and are retained for backwards compatibility.

With the new hook management several additional hooks have been added to LaTeX and more will follow. See the next section for what is already available.

\AtBeginDocument `\AtBeginDocument [⟨label⟩] {⟨code⟩}`

If used without the optional argument ⟨*label*⟩, it works essentially like before, i.e., it is adding ⟨*code*⟩ to the hook `begindocument` (which is executed inside `\begin{document}`). However, all code added this way is labeled with the label `top-level` if done outside of a package or class or with the package/class name if called inside such a file.

This way one can add further code to the hook using `\AddToHook` or `\AtBeginDocument` using a different label and explicitly order the code chunks as necessary, e.g., run some code before or after the `top-level` code. When using the optional argument the call is equivalent to running `\AddToHook {begindocument} [`⟨*label*⟩`] {`⟨*code*⟩`}`.

For important packages with known order requirement we may over time add rules to the kernel (or to those packages) so that they work regardless of the loading-order in the document.

\AtEndDocument `\AtEndDocument [⟨label⟩] {⟨code⟩}`

Like `\AtBeginDocument` but for the `enddocument` hook.

\AtBeginDvi `\AtBeginDvi [⟨label⟩] {⟨code⟩}`

This hook is discussed in conjunction with the shipout hooks.

2.7 LaTeX 2ε commands and environments augmented by hooks

intro to be written

2.7.1 Generic hooks for all environments

Every environment ⟨*env*⟩ has now four associated hooks coming with it:

env/⟨*env*⟩/before This hook is executed as part of `\begin` as the very first action, in particular prior to starting the environment group. Its scope is therefore not restricted by the environment.

env/⟨*env*⟩/begin This hook is executed as part of `\begin` directly in front of the code specific to the environment start (e.g., the second argument of `\newenvironment`). Its scope is the environment body.

env/⟨*env*⟩/end This is executed as part of `\end` directly in front of the code specific to the end of the environment (e.g., the third argument of `\newenvironment`).

easy to find out how the kernel adds to them. The main reason of this section is therefore to say "please don't do that, this is unconfigurable code!"

env/⟨*env*⟩/after This is executed as part of `\end` after the code specific to the environment end and after the environment group has ended. Its scope is therefore not restricted by the environment.

This hook is implemented as a reversed hook so if two packages add code to `env/⟨env⟩/before` and to `env/⟨env⟩/after` they can add surrounding environments and the order of closing them happens in the right sequence.

In contrast to other hooks these hooks do not need to be declared using `\NewHook`.

The hooks are only executed if `\begin{⟨env⟩}` and `\end{⟨env⟩}` is used. If the environment code is executed via low-level calls to `\⟨env⟩` and `\end⟨env⟩` (e.g., to avoid the environment grouping) they are not available. If you want them available in code using this method, you would need to add them yourself, i.e., write something like

```
\UseHook{env/quote/before}\quote
    ...
\endquote\UseHook{env/quote/after}
```

to add the outer hooks, etc.

2.7.2 Hooks provided by `\begin{document}`

Until 2020 `\begin{document}` offered exactly one hook that one had to fill using `\AtBeginDocument`. Experience has shown that this single hook in one place was not enough and as part of adding the general hook management system a number of additional hooks have been added at this point. The places for hooks have been chosen to provide the same support as offered by external packages, such as etoolbox and others that augmented `\document` to gain better control.

Supported are now the following hooks:

env/document/before This is the generic environment hook executed effectively before `\begin{document}` starts, i.e., one can think of it as a hook for code at the end of the preamble section.

env/document/begin This is the second generic environment hook that is executed after the environment has started its group. But given that for the `document` environment this group is canceled there is little difference to the previous one as the two are directly executed one after another (the only difference is that in this hook `\@currenvir` is now set to `document` but anybody adding to this hook would know that already).

begindocument This hook is added to by `\AtBeginDocument` and is executed after the `.aux` file as be read in and most initialization are done, so they can be altered and inspected by the hook code. It is followed by a small number of further initializations that shouldn't be altered and are therefore coming later.

begindocument/end This hook is executed at the end of the `\document` code in other words at the beginning of the document body. The only command that follows it is `\ignorespaces`.

2.7.3 Hooks provided by `\end{document}`

LaTeX 2$_\varepsilon$ always provided `\AtEndDocument` to add code to the execution of `\end{document}` just in front of the code that is normally executed there. While this was a big improvement over the situation in LaTeX 2.09 it was not flexible enough for a number of use cases and so packages, such as etoolbox, atveryend and others patched `\enddocument` to add additional points where code could be hooked into.

Patching using packages is always problematical as leads to conflicts (code availability, ordering of patches, incompatible patches, etc.). For this reason a number of additional hooks have been added to the `\enddocument` code to allow packages to add code in various places in a controlled way without the need for overwriting or patching the core code.

Supported are now the following hooks:

`env/document/end` The generic hook inside `\end`.

`enddocument` The hook associated with `\AtEndDocument`. It is immediately called after the previous hook so there could be just one.[6]

When this hook is executed there may be still unprocessed material (e.g., floats on the deferlist) and the hook may add further material to be typeset. After it, `\clearpage` is called to ensure that all such material gets typeset. If there is nothing waiting the `\clearpage` has no effect.

`enddocument/afterlastpage` As the name indicates this hook should not receive code that generates material for further pages. It is the right place to do some final housekeeping and possibly write out some information to the `.aux` file (which is still open at this point to receive data). It is also the correct place to set up any testing code to be run when the `.aux` file is re-read in the next step.

After this hook has been executed the `.aux` file is closed for writing and then read back in to do some tests (e.g., looking for missing references or duplicated labels, etc.).

`enddocument/afteraux` At this point, the `.aux` file has been reprocessed and so this is a possible place for final checks and display of information to the user. However, for the latter you might prefer the next hook, so that your information is displayed after the (possibly longish) list of files if that got requested via `\listfiles`.

`enddocument/info` This hook is meant to receive code that write final information messages to the terminal. It follows immediately after the previous hook (so both could have been combined, but then packages adding further code would always need to also supply an explicit rule to specify where it should go.

This hook already contains some code added by the kernel (under the labels `kernel/filelist` and `kernel/warnings`), namely the list of files when `\listfiles` has been used and the warnings for duplicate labels, missing references, font substitutions etc.

`enddocument/end` Finally, this hook is executed just in front of the final call to `\@@end`.

[6] We could make `\AtEndDocument` just fill the `env/document/end` but maybe that is a bit confusing.

There is also the hook `shipout/lastpage`. This hook is executed as part of the last `\shipout` in the document to allow package to add final `\special`'s to that page. Where this hook is executed in relation to those from the above list can vary from document to document. Furthermore to determine correctly which of the `\shipouts` is the last one, LaTeX needs to be run several times, so initially it might get executed on the wrong page. See section 2.7.4 for where to find the details.

2.7.4 Hooks provided `\shipout` operations

There are several hooks and mechanisms added to LaTeX's process of generating pages. These are documented in `ltshipout-doc.pdf` or with code in `ltshipout-code.pdf`.

2.7.5 Hooks provided file loading operations

There are several hooks added to LaTeX's process of loading file via its high-level interfaces such as `\input`, `\include`, `\usepackage`, etc. These are documented in `ltfilehook-doc.pdf` or with code in `ltfilehook-code.pdf`.

3 The Implementation

```
⟨@@=hook⟩
⟨*2ekernel⟩
\ExplSyntaxOn
```

3.1 Debugging

`\g__hook_debug_bool` Holds the current debugging state.

```
\bool_new:N \g__hook_debug_bool
```

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

`\hook_debug_on:` `\hook_debug_off:` `\__hook_debug:n` `\__hook_debug_gset:` Turns debugging on and off by redefining `\__hook_debug:n`.

```
\cs_new_eq:NN \__hook_debug:n \use_none:n
\cs_new_protected:Npn \hook_debug_on:
  {
    \bool_gset_true:N \g__hook_debug_bool
    \__hook_debug_gset:
  }
\cs_new_protected:Npn \hook_debug_off:
  {
    \bool_gset_false:N \g__hook_debug_bool
    \__hook_debug_gset:
  }
\cs_new_protected:Npn \__hook_debug_gset:
  {
    \cs_gset_protected:Npx \__hook_debug:n ##1
      { \bool_if:NT \g__hook_debug_bool {##1} }
  }
```

(*End definition for* `\hook_debug_on:` *and others. These functions are documented on page* 11.)

3.2 Borrowing from internals of other kernel modules

__hook_str_compare:nn Private copy of __str_if_eq:nn

```
\cs_new_eq:NN \__hook_str_compare:nn \__str_if_eq:nn
```

(*End definition for* __hook_str_compare:nn.)

3.3 Declarations

\l__hook_return_tl Scratch variables used throughout the package.
\l__hook_tmpa_tl
\l__hook_tmpb_tl

```
\tl_new:N \l__hook_return_tl
\tl_new:N \l__hook_tmpa_tl
\tl_new:N \l__hook_tmpb_tl
```

(*End definition for* \l__hook_return_tl, \l__hook_tmpa_tl, *and* \l__hook_tmpb_tl.)

\g__hook_all_seq In a few places we need a list of all hook names ever defined so we keep track if them in this sequence.

```
\seq_new:N \g__hook_all_seq
```

(*End definition for* \g__hook_all_seq.)

\g__hook_removal_list_prop A property list to hold delayed removals.

```
\tl_new:N \g__hook_removal_list_tl
```

(*End definition for* \g__hook_removal_list_prop.)

\l__hook_cur_hook_tl Stores the name of the hook currently being sorted.

```
\tl_new:N \l__hook_cur_hook_tl
```

(*End definition for* \l__hook_cur_hook_tl.)

\g__hook_code_temp_prop A property list to temporarily save the original one so that it isn't permanently changed during sorting.

```
\prop_new:N \g__hook_code_temp_prop
```

(*End definition for* \g__hook_code_temp_prop.)

\g__hook_hook_curr_name_tl Default label used for hook commands, and a stack to keep track of packages within packages.
\g__hook_name_stack_seq

```
\tl_new:N \g__hook_hook_curr_name_tl
\seq_new:N \g__hook_name_stack_seq
```

(*End definition for* \g__hook_hook_curr_name_tl *and* \g__hook_name_stack_seq.)

__hook_tmp:w Temporary macro for generic usage.

```
\cs_new_eq:NN \__hook_tmp:w ?
```

(*End definition for* __hook_tmp:w.)

\tl_gremove_once:Nx Some variants of expl3 functions.

FMi: should be moved to expl3

```
\cs_generate_variant:Nn \tl_gremove_once:Nn { Nx }
```

(*End definition for* \tl_gremove_once:Nx. *This function is documented on page* ??.)

\s__hook_mark Scan mark used for delimited arguments.

```
\scan_new:N \s__hook_mark
```

(*End definition for* \s__hook_mark.)

3.4 Providing new hooks

\g__hook_..._code_prop
\g__hook_..._rules_prop
\g__hook_..._code_tl
\g__hook_..._next_code_tl

Hooks have a ⟨*name*⟩ and for each hook we have to provide a number of data structures. These are

`\g__hook_`⟨*name*⟩`_code_prop` A property list holding the code for the hook in separate chunks. The keys are by default the package names that add code to the hook, but it is possible for packages to define other keys.

`\g__hook_`⟨*name*⟩`_rules_prop` A property listing holding relation info how the code chunks should be ordered within a hook. This is used for debugging only. The actual rule for a ⟨*hook*⟩ is stored in a separate token lists named `\g__hook_`⟨*hook*⟩`_rule_`⟨*label1*⟩`|`⟨*label2*⟩`_tl` for a pair of labels.

`\g__hook_`⟨*name*⟩`_code_tl` The code that is actually executed when the hook is called in the document is stored in this token list. It is constructed from the code chunks applying the information.

`\g__hook_`⟨*name*⟩`_next_code_tl` Finally there is extra code (normally empty) that is used on the next invocation of the hook (and then deleted). This can be used to define some special behavior for a single occasion from within the document.

(*End definition for* `\g__hook_..._code_prop` *and others.*)

\hook_new:n

The `\hook_new:n` declaration declare a new hook and expects the hook ⟨*name*⟩ as its argument, e.g., `begindocument`.

```
\cs_new_protected:Npn \hook_new:n #1
  {
    \exp_args:Nx \__hook_new:n
      { \__hook_parse_label_default:nn {#1} { top-level } }
  }
\cs_new_protected:Npn \__hook_new:n #1 {
```

We check for one of the internal data structures and if it already exists we complain.

```
  \hook_if_exist:nTF {#1}
    { \ErrorHookExists }
```

Otherwise we add the hook name to the list of all hooks and allocate the necessary data structures for the new hook.

```
    { \seq_gput_right:Nn \g__hook_all_seq {#1}
```

This is only used by the actual code of the current hook, so declare it normally:

```
      \tl_new:c { g__hook_#1_code_tl }
```

Now ensure that the base data structure for the hook exists:

```
      \__hook_declare:n {#1}
```

The `\g__hook_`⟨*hook*⟩`_labels_clist` holds the sorted list of labels (once it got sorted). This is used only for debugging.

```
      \clist_new:c {g__hook_#1_labels_clist}
```

Some hooks should reverse the default order of code chunks. To signal this we have a token list which is empty for normal hooks and contains a `-` for reversed hooks.

```
      \tl_new:c { g__hook_#1_reversed_tl }
```

The above is all in L3 convention, but we also provide an interface to legacy LaTeX 2$_\varepsilon$ for use in the current kernel. This is done in a separate macro.

```
        \__hook_provide_legacy_interface:n {#1}
      }
}
```

(*End definition for* `\hook_new:n`. *This function is documented on page 9.*)

`\__hook_declare:n` This function declares the basic data structures for a hook without actually declaring the hook itself. This is needed to allow adding to undeclared hooks. Here it is unnecessary to check whether both variables exist, since both are declared at the same time (either both exist, or neither).

```
\cs_new_protected:Npn \__hook_declare:n #1
  {
    \__hook_if_exist:nF {#1}
      {
        \prop_new:c { g__hook_#1_code_prop }
        \tl_new:c { g__hook_#1_next_code_tl }
        \prop_new:c { g__hook_#1_rules_prop } % only for debugging
      }
  }
```

(*End definition for* `\__hook_declare:n`.)

`\hook_new_reversed:n` Declare a new hook. The default ordering of code chunks is reversed, signaled by setting the token list to a minus sign.

```
\cs_new_protected:Npn \hook_new_reversed:n #1 {
  \hook_new:n {#1}
  \tl_gset:cn { g__hook_#1_reversed_tl } { - }
}
```

(*End definition for* `\hook_new_reversed:n`. *This function is documented on page 9.*)

`\hook_new_pair:nn` A shorthand for declaring a normal and a (matching) reversed hook in one go.

```
\cs_new_protected:Npn \hook_new_pair:nn #1#2 {
  \hook_new:n {#1}  \hook_new_reversed:n {#2}
}
```

(*End definition for* `\hook_new_pair:nn`. *This function is documented on page 9.*)

`\__hook_provide_legacy_interface:n` The LaTeX legacy concept for hooks uses with hooks the following naming scheme in the code: `\@...hook`.

We follow this convention and insert the hook code using this naming scheme in LaTeX 2$_\varepsilon$. At least as long as this code is in a package, some such hooks are already filled with data when we move them over to the new scheme. We therefore insert already existing code under the label `legacy` into the hook management machinery and then replace the `\@...hook` with its counterpart which is `\g__hook_#1_code_tl`.[7]

```
\cs_new_protected:Npn \__hook_provide_legacy_interface:n #1
  {
```

[7] This means one extra unnecessary expansion on each invocation in the document but keeps the LaTeX 2$_\varepsilon$ and the L3 coding side properly separated.

If the expl3 code is run with checking on then assigning or using non L3 names such as `\@enddocumenthook` with expl3 functions will trigger warnings so we run this code with debugging explicitly suspended.

```
    \debug_suspend:
    \tl_if_exist:cT { @#1hook }
```

Of course if the hook exists but is still empty, there is no need to add anything under `legacy` or the current package name.

```
      {
        \tl_if_empty:cF { @#1hook }
          {
            \__hook_gput_code:nxv {#1}
              { \__hook_parse_label_default:Vn \c_novalue_tl { legacy } }
              { @#1hook }
          }
      }
```

We need a global definition in case the declaration is done inside a group (which happens below at the end of the file). This is another reason why need to suspend checking, otherwise `\tl_gset:co` would complain about `\@...hook` not starting with `\g_`.

```
    \tl_gset:co{@#1hook}{\cs:w g__hook_#1_code_tl\cs_end:}
    \debug_resume:
  }
```

(*End definition for* `\__hook_provide_legacy_interface:n`.)

3.5 Parsing a label

`\__hook_parse_label_default:nn`
`\__hook_parse_label_default:Vn`

This macro checks if a label was given (not `\c_novalue_tl`), and if so, tries to parse the label looking for a leading `.` to replace for `\@currname`. Otherwise `\__hook_currname_or_default:n` is used to pick `\@currname` or the fallback value.

```
\cs_new:Npn \__hook_parse_label_default:nn #1 #2
  {
    \tl_if_novalue:nTF {#1}
      { \__hook_currname_or_default:n {#2} }
      { \tl_trim_spaces_apply:nN {#1} \__hook_parse_dot_label:nn {#2} }
  }
\cs_generate_variant:Nn \__hook_parse_label_default:nn { V }
```

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

`\__hook_parse_dot_label:nn`
`\__hook_parse_dot_label:nw`
`\__hook_parse_dot_label_cleanup:w`
`\__hook_parse_dot_label_aux:nw`

Start by checking if the label is empty, which raises an error, and uses the fallback value. If not, split the label at a `./`, if any, and check if no tokens are before the `./`, or if the only character is a `.`. If these requirements are fulfilled, the leading `.` is replaced with `\__hook_currname_or_default:n`. Otherwise the label is returned unchanged.

```
\cs_new:Npn \__hook_parse_dot_label:nn #1 #2
  {
    \tl_if_empty:nTF {#1}
      {
        \msg_expandable_error:nnn { hooks } { empty-label } {#2}
        #2
      }
      {
        \str_if_eq:nnTF {#1} { . }
```

```
          { \__hook_currname_or_default:n {#1} }
          { \__hook_parse_dot_label:nw {#2} #1 ./ \s__hook_mark }
      }
  }
\cs_new:Npn \__hook_parse_dot_label:nw #1 #2 ./ #3 \s__hook_mark
  {
    \tl_if_empty:nTF {#2}
      { \__hook_parse_dot_label_aux:nw {#1} #3 \s__hook_mark }
      {
        \tl_if_empty:nTF {#3}
          {#2}
          { \__hook_parse_dot_label_cleanup:w #2 ./ #3 \s__hook_mark }
      }
  }
\cs_new:Npn \__hook_parse_dot_label_cleanup:w #1 ./ \s__hook_mark {#1}
\cs_new:Npn \__hook_parse_dot_label_aux:nw #1 #2 ./ \s__hook_mark
  { \__hook_currname_or_default:n {#1} / #2 }
```

(*End definition for* `\__hook_parse_dot_label:nn` *and others.*)

`\__hook_currname_or_default:n`

Uses `\g__hook_hook_curr_name_tl` if it is set, otherwise tries `\@currname`. If neither is set, uses the fallback value `#1` (usually `top-level`).

```
\cs_new:Npn \__hook_currname_or_default:n #1
  {
    \tl_if_empty:NTF \g__hook_hook_curr_name_tl
      {
        \tl_if_empty:NTF \@currname
          {#1}
          { \@currname }
      }
      { \g__hook_hook_curr_name_tl }
  }
```

(*End definition for* `\__hook_currname_or_default:n`.)

`\hook_gput_code:nnn`
`\__hook_gput_code:nnn`
`\__hook_gput_code:nxv`
`\__hook_hook_gput_code_do:nnn`

With `\hook_gput_code:nnn{`⟨*hook*⟩`}{`⟨*label*⟩`}{`⟨*code*⟩`}` a chunk of ⟨*code*⟩ is added to an existing ⟨*hook*⟩ labeled with ⟨*label*⟩.

```
\cs_new_protected:Npn \hook_gput_code:nnn #1 #2
  {
    \exp_args:Nxx \__hook_gput_code:nnn
      { \__hook_parse_label_default:nn {#1} { top-level } }
      { \__hook_parse_label_default:nn {#2} { top-level } }
  }
\cs_new_protected:Npn \__hook_gput_code:nnn #1 #2 #3
  {
```

First we check if the hook exists.

```
    \__hook_if_marked_removal:nnTF {#1} {#2}
      { \__hook_unmark_removal:nn {#1} {#2} }
      {
```

First we check if the hook exists.

```
        \hook_if_exist:nTF {#1}
```

If so we simply add (or append) the new code to the property list holding different chunks for the hook. At `\begin{document}` this is then sorted into a token list for fast execution.

```
          {
            \__hook_hook_gput_code_do:nnn {#1} {#2} {#3}
```

However, if there is an update within the document we need to alter this execution code which is done by `\__hook_update_hook_code:n`. In the preamble this does nothing.

```
            \__hook_update_hook_code:n {#1}
          }
          { \__hook_try_declaring_generic_hook:nnn {#1} {#2} {#3} }
      }
  }
\cs_generate_variant:Nn \__hook_gput_code:nnn { nxv }
```

This macro will unconditionally add a chunk of code to the given hook.

```
\cs_new_protected:Npn \__hook_hook_gput_code_do:nnn #1 #2 #3
  {
%   However, first some debugging info if debugging is enabled:
%    \begin{macrocode}
    \__hook_debug:n{\iow_term:x{****~ Add~ to~
                        \hook_if_exist:nF {#1} { undeclared~ }
                        hook~ #1~ (#2)
                        \on@line\space <-~ \tl_to_str:n{#3}} }
```

Then try to get the code chunk labeled `#2` from the hook. If there's code already there, then append `#3` to that, otherwise just put `#3`.

```
    \prop_get:cnNTF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
      {
        \prop_gput:cno { g__hook_#1_code_prop } {#2}
          { \l__hook_return_tl #3 }
      }
      { \prop_gput:cnn { g__hook_#1_code_prop } {#2} {#3} }
  }
```

(*End definition for* `\hook_gput_code:nnn`*,* `\__hook_gput_code:nnn`*, and* `\__hook_hook_gput_code_do:nnn`*. This function is documented on page* 10*.*)

`\__hook_gput_undeclared_hook:nnn`

Often it may happen that a package A defines a hook `foo`, but package B, that adds code to that hook, is loaded before A. In such case we need to add code to the hook before its declared.

```
\cs_new_protected:Npn \__hook_gput_undeclared_hook:nnn #1 #2 #3
  {
    \__hook_declare:n {#1}
    \__hook_hook_gput_code_do:nnn {#1} {#2} {#3}
  }
```

(*End definition for* `\__hook_gput_undeclared_hook:nnn`*.*)

`\__hook_try_declaring_generic_hook:nnn`
`\__hook_try_declaring_generic_next_hook:nn`

These entry-level macros just pass the arguments along to the common `\__hook_try_declaring_generic_hook:nNNnn` with the right functions to execute when some action is to be taken.

The wrapper `\__hook_try_declaring_generic_hook:nnn` then defers `\hook_gput_code:nnn` if the generic hook was declared, or to `\__hook_gput_undeclared_hook:nnn` otherwise (the hook was tested for existence before, so at this point if it isn't generic, it doesn't exist).

The wrapper `\__hook_try_declaring_generic_next_hook:nn` for next-execution hooks does the same: it defers the code to `\hook_gput_next_code:nn` if the generic hook was declared, or to `\__hook_gput_next_do:nn` otherwise.

```
\cs_new_protected:Npn \__hook_try_declaring_generic_hook:nnn #1
  {
    \__hook_try_declaring_generic_hook:nNNnn {#1}
      \hook_gput_code:nnn \__hook_gput_undeclared_hook:nnn
  }
\cs_new_protected:Npn \__hook_try_declaring_generic_next_hook:nn #1
  {
    \__hook_try_declaring_generic_hook:nNNnn {#1}
      \hook_gput_next_code:nn \__hook_gput_next_do:nn
  }
```

__hook_try_declaring_generic_hook:nNNnn
__hook_try_declaring_generic_hook_split:nNNnn
__hook_try_declaring_generic_hook:wn*TF*

`\__hook_try_declaring_generic_hook:nNNnn` now splits the hook name at the first `/` (if any) and first checks if it is a file-specific hook (they require some normalization) using `\__hook_if_file_hook:wTF`. If not then check it is one of a predefined set for generic names. We also split off the second component to see if we have to make a reversed hook. In either case the function returns ⟨*true*⟩ for a generic hook and ⟨*false*⟩ in other cases.

```
\cs_new_protected:Npn \__hook_try_declaring_generic_hook:nNNnn #1
  {
    \__hook_if_file_hook:wTF #1 / / \s__hook_mark
      {
        \exp_args:Ne \__hook_try_declaring_generic_hook_split:nNNnn
          { \exp_args:Ne \__hook_file_hook_normalise:n {#1} }
      }
      { \__hook_try_declaring_generic_hook_split:nNNnn {#1} }
  }
\cs_new_protected:Npn \__hook_try_declaring_generic_hook_split:nNNnn #1 #2 #3
  {
    \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
      { #2 }
      { #3 } {#1}
  }
\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
    #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
  {
    \tl_if_empty:nTF {#2}
      { \prg_return_false: }
      {
        \prop_if_in:NnTF \c__hook_generics_prop {#1}
          {
            \hook_if_exist:nF {#5} { \hook_new:n {#5} }
```

After having declared the hook we check the second component (for file hooks) or the third component for environment hooks) and if it is on the list of components for which we should have declared a reversed hook we alter the hook data structure accordingly.

```
            \prop_if_in:NnTF \c__hook_generics_reversed_ii_prop {#2}
              { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
              {
                \prop_if_in:NnT \c__hook_generics_reversed_iii_prop {#3}
                  { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
              }
```

Now that we know that the hook is declared we can add the code to it.

```
            \prg_return_true:
          }
          { \prg_return_false: }
      }
  }
```

(*End definition for* `\__hook_try_declaring_generic_hook:nnn` *and others.*)

`\__hook_if_file_hook_p:w`
`\__hook_if_file_hook:wTF`

`\__hook_if_file_hook:wTF` checks if the argument is a valid file-specific hook (not, for example, `file/before`, but `file/before/foo.tex`). If it is a file-specific hook, then it executes the ⟨*true*⟩ branch, otherwise ⟨*false*⟩.

A file-specific hook is `file/`⟨*position*⟩`/`⟨*name*⟩. If any of these parts don't exist, it is a general file hook or not a file hook at all, so the conditional evaluates to ⟨*false*⟩. Otherwise, it checks that the first part is `file` and that the ⟨*position*⟩ is in the `\c__hook_generics_file_prop`.

A property list is used here to avoid having to worry with catcodes, because expl3's file name parsing turns all characters into catcode-12 tokens, which might differ from hand-input letters.

```
\prg_new_conditional:Npnn \__hook_if_file_hook:w
    #1 / #2 / #3 \s__hook_mark { TF }
  {
    \str_if_eq:nnTF {#1} { file }
      {
        \bool_lazy_or:nnTF
            { \tl_if_empty_p:n {#3} }
            { \str_if_eq_p:nn {#3} { / } }
          { \prg_return_false: }
          {
            \prop_if_in:NnTF \c__hook_generics_file_prop {#2}
              { \prg_return_true: }
              { \prg_return_false: }
          }
      }
      { \prg_return_false: }
  }
```

(*End definition for* `\__hook_if_file_hook:wTF`.)

`\__hook_file_hook_normalise:n`
`\__hook_strip_double_slash:n`
`\__hook_strip_double_slash:w`

When a file-specific hook is found, before being declared it is lightly normalized by `\__hook_file_hook_normalise:n`. The current implementation just replaces two consecutive slashes (`//`) by a single one, to cope with simple cases where the user did something like `\def\input@path{{./mypath/}}`, in which case a hook would have to be `\AddToHook{file/after/./mypath//file.tex}`.

```
\cs_new:Npn \__hook_file_hook_normalise:n #1
  { \__hook_strip_double_slash:n {#1} }
\cs_new:Npn \__hook_strip_double_slash:n #1
  { \__hook_strip_double_slash:w #1 // \s__hook_mark }
\cs_new:Npn \__hook_strip_double_slash:w #1 // #2 \s__hook_mark
  {
    \tl_if_empty:nTF {#2}
      {#1}
      { \__hook_strip_double_slash:w #1 / #2 \s__hook_mark }
  }
```

(*End definition for* `\__hook_file_hook_normalise:n`, `\__hook_strip_double_slash:n`, *and* `\__hook_strip_double_slash:w`.)

`\c__hook_generics_prop` Clist holding the generic names. We don't provide any user interface to this as this is meant to be static.

`env` The generic hooks used in `\begin` and `\end`.

`file` The generic hooks used when loading a file

```
\prop_const_from_keyval:Nn \c__hook_generics_prop
  {env=,file=,package=,class=,include=}
```

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

`\c__hook_generics_reversed_ii_prop` `\c__hook_generics_reversed_iii_prop` `\c__hook_generics_file_prop` Some of the generic hooks are supposed to use reverse ordering, these are the following (only the second or third sub-component is checked):

```
\prop_const_from_keyval:Nn \c__hook_generics_reversed_ii_prop {after=,end=}
\prop_const_from_keyval:Nn \c__hook_generics_reversed_iii_prop {after=}
\prop_const_from_keyval:Nn \c__hook_generics_file_prop {before=,after=}
```

(*End definition for* `\c__hook_generics_reversed_ii_prop`, `\c__hook_generics_reversed_iii_prop`, *and* `\c__hook_generics_file_prop`.)

`\__hook_update_hook_code:n` Before `\begin{document}` this does nothing, in the body it reinitializes the hook code using the altered data.

```
\cs_new_eq:NN \__hook_update_hook_code:n \use_none:n
```

(*End definition for* `\__hook_update_hook_code:n`.)

`\hook_gremove_code:nn` `\__hook_gremove_code:nn` With `\hook_gremove_code:nn{`⟨*hook*⟩`}{`⟨*label*⟩`}` any code for ⟨*hook*⟩ stored under ⟨*label*⟩ is removed.

```
\cs_new_protected:Npn \hook_gremove_code:nn #1 #2
  {
    \exp_args:Nxx \__hook_gremove_code:nn
      { \__hook_parse_label_default:nn {#1} { top-level } }
      { \__hook_parse_label_default:nn {#2} { top-level } }
  }
\cs_new_protected:Npn \__hook_gremove_code:nn #1 #2
  {
```

First check that the hook code pool exists. `\hook_if_exist:nTF` isn't used here because it should be possible to remove code from a hook before its defined (see section 2.1.5).

```
    \__hook_if_exist:nTF {#1}
```

Then remove the chunk and run `\__hook_update_hook_code:n` so that the execution token list reflects the change if we are after `\begin{document}`.

```
      {
        \str_if_eq:nnTF {#2} {*}
          {
            \prop_gclear:c { g__hook_#1_code_prop }
            \clist_gclear:c { g__hook_#1_labels_clist } % for debugging only
          }
          {
```

Check if the label being removed exists in the code pool. If it does, just call `\__hook_gremove_code_do:nn` to do the removal, otherwise mark it to be removed.

```
            \prop_get:cnNTF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
              { \__hook_gremove_code_do:nn }
              { \__hook_mark_removal:nn }
                  {#1} {#2}
          }
```

Finally update the code, if the hook exists.

```
        \hook_if_exist:nT {#1}
          { \__hook_update_hook_code:n {#1} }
      }
      { \__hook_mark_removal:nn {#1} {#2} }
  }
```

`\__hook_gremove_code_do:nn`

```
\cs_new_protected:Npn \__hook_gremove_code_do:nn #1 #2
  {
    \prop_gremove:cn { g__hook_#1_code_prop } {#2}
```

Removing the dropped label from `\g_@@_#1_labels_clist` is rather tricky, because that clists holds the labels as strings (i.e., not ordinary text which is what we have in `#2`).

```
    \exp_args:Nco \clist_gremove_all:Nn
      { g__hook_#1_labels_clist } { \tl_to_str:n {#2} } % for debugging only
  }
```

(*End definition for* `\hook_gremove_code:nn`*,* `\__hook_gremove_code:nn`*, and* `\__hook_gremove_code_do:nn`*. This function is documented on page 10.*)

`\__hook_mark_removal:nn` Marks ⟨*label*⟩ (`#2`) to be removed from ⟨*hook*⟩ (`#1`).

```
\cs_new_protected:Npn \__hook_mark_removal:nn #1 #2
  {
    \tl_gput_right:Nx \g__hook_removal_list_tl
      { \__hook_removal_tl:nn {#1} {#2} }
  }
```

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

`\__hook_unmark_removal:nn` Unmarks ⟨*label*⟩ (`#2`) to be removed from ⟨*hook*⟩ (`#1`). `\tl_gremove_once:Nx` is used rather than `\tl_gremove_all:Nx` so that two additions are needed to cancel two marked removals, rather than only one.

```
\cs_new_protected:Npn \__hook_unmark_removal:nn #1 #2
  {
    \tl_gremove_once:Nx \g__hook_removal_list_tl
      { \__hook_removal_tl:nn {#1} {#2} }
  }
```

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

`__hook_if_marked_removal:nn`*TF* Checks if the `\g__hook_removal_list_tl` contains the current ⟨*label*⟩ (`#2`) and ⟨*hook*⟩ (`#1`).

```
\prg_new_protected_conditional:Npnn \__hook_if_marked_removal:nn #1 #2 { TF }
  {
    \exp_args:NNx \tl_if_in:NnTF \g__hook_removal_list_tl
      { \__hook_removal_tl:nn {#1} {#2} }
      { \prg_return_true: } { \prg_return_false: }
  }
```

(*End definition for* `\__hook_if_marked_removal:nnTF`.)

`\__hook_removal_tl:nn` Builds a token list with `#1` and `#2` which can only be matched by `#1` and `#2`.

```
\cs_new:Npn \__hook_removal_tl:nn #1 #2
  { & \tl_to_str:n {#2} $ \tl_to_str:n {#1} $ }
```

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

`\g__hook_??_rules_prop` `\g__hook_??_code_prop` `\g__hook_??_code_tl` `\g__hook_??_reversed_tl`

Default rules applying to all hooks are stored in this property list. Initially it simply used an empty "label" name (not two question marks). This was a bit unfortunate, because then `l3doc` complains about `__` in the middle of a command name when trying to typeset the documentation. However using a "normal" name such as `default` has the disadvantage of that being not really distinguishable from a real hook name. I now have settled for `??` which needs some gymnastics to get it into the csname, but since this is used a lot things should be fast, so this is not done with `c` expansion in the code later on.

`\g__hook_??_code_tl` isn't used, but it has to be defined to trick the code into thinking that `??` is actually a hook.

```
\prop_new:c {g__hook_??_rules_prop}
\prop_new:c {g__hook_??_code_prop}
\prop_new:c {g__hook_??_code_tl}
```

Default rules are always given in normal ordering (never in reversed ordering). If such a rule is applied to a reversed hook it behaves as if the rule is reversed (e.g., `after` becomes `before`) because those rules are applied first and then the order is reversed.

```
\tl_new:c {g__hook_??_reversed_tl}
```

(*End definition for* `\g__hook_??_rules_prop` *and others.*)

`\__hook_debug_gset_rule:nnnn`

FMi: this needs cleanup and docu correction!

With `\__hook_debug_gset_rule:nnnn{`⟨*hook*⟩`}{`⟨*label1*⟩`}{`⟨*relation*⟩`}{`⟨*label2*⟩`}` a relation is defined between the two code labels for the given ⟨*hook*⟩. The special hook `??` stands for *any* hook describing a default rule.

```
\cs_new_protected:Npn \__hook_debug_gset_rule:nnnn #1#2#3#4
  {
```

If so we drop any existing rules with the two labels (in case there are any).

```
    \prop_gremove:cn{g__hook_#1_rules_prop}{#2|#4}
    \prop_gremove:cn{g__hook_#1_rules_prop}{#4|#2}
```

Then we add the new one (normalizing the input a bit, e.g., we always use `before` and not `after` and instead reorder the labels):

```
    \str_case_e:nnF {#3}
       {
         {before} { \prop_gput:cnn {g__hook_#1_rules_prop}{#2|#4}{<} }
         {after}  { \prop_gput:cnn {g__hook_#1_rules_prop}{#4|#2}{<} }
```

More special rule types ...

```
         {incompatible-error}   { \prop_gput:cnn {g__hook_#1_rules_prop}{#2|#4}{xE} }
         {incompatible-warning} { \prop_gput:cnn {g__hook_#1_rules_prop}{#2|#4}{xW} }
         {removes}      { \prop_gput:cnn {g__hook_#1_rules_prop}{#2|#4}{->} }
```

Undo a setting:

```
          {unrelated}{ \prop_gremove:cn {g__hook_#1_rules_prop}{#2|#4}
                       \prop_gremove:cn {g__hook_#1_rules_prop}{#4|#2} }
        }
        { \ERROROunknownrule }
    }
```

(*End definition for* `\__hook_debug_gset_rule:nnnn`.)

3.6 Setting rules for hooks code

`\hook_gset_rule:nnnn`
`\__hook_gset_rule:nnnn`

FMi: needs docu correction given new implementation

With `\hook_gset_rule:nnnn{`⟨*hook*⟩`}{`⟨*label1*⟩`}{`⟨*relation*⟩`}{`⟨*label2*⟩`}` a relation is defined between the two code labels for the given ⟨*hook*⟩. The special hook `??` stands for *any* hook describing a default rule.

```
\cs_new_protected:Npn \hook_gset_rule:nnnn #1#2#3#4
  {
    \use:x
      {
        \__hook_gset_rule:nnnn
          { \__hook_parse_label_default:nn {#1} { top-level } }
          { \__hook_parse_label_default:nn {#2} { top-level } }
          {#3}
          { \__hook_parse_label_default:nn {#4} { top-level } }
      }
  }
\cs_new_protected:Npn \__hook_gset_rule:nnnn #1#2#3#4
  {
```

First we ensure the basic data structure of the hook exists:

```
    \__hook_declare:n {#1}
```

Then we clear any previous relationship between both labels.

```
    \__hook_rule_gclear:nnn {#1} {#2} {#4}
```

Then we call the function to handle the given rule. Throw an error if the rule is invalid.

```
    \debug_suspend:
    \cs_if_exist_use:cTF { __hook_rule_#3_gset:nnn }
      {
          {#1} {#2} {#4}
        \__hook_update_hook_code:n {#1}
      }
      { \ERROROunknownrule }
    \debug_resume:
    \__hook_debug_gset_rule:nnnn {#1} {#2} {#3} {#4} % for debugging
  }
```

(*End definition for* `\hook_gset_rule:nnnn` *and* `\__hook_gset_rule:nnnn`. *This function is documented on page* 10.)

`\__hook_rule_before_gset:nnn`
`\__hook_rule_after_gset:nnn`
`\__hook_rule_<_gset:nnn`
`\__hook_rule_>_gset:nnn`

Then we add the new rule. We need to normalize the rules here to allow for faster processing later. Given a pair of labels l_A and l_B, the rule $l_A > l_B$ is the same as $l_B < l_A$

FMi:

said differently. But normalizing the forms of the rule to a single representation, say, $l_B < l_A$, then the time spent looking for the rules later is considerably reduced.

Here we do that normalization by using `\(pdf)strcmp` to lexically sort labels l_A and l_B to a fixed order. This order is then enforced every time these two labels are used together.

Here we use `\__hook_label_pair:nn {`⟨*hook*⟩`} {`⟨l_A⟩`} {`⟨l_B⟩`}` to build a string $l_B \mid l_A$ with a fixed order, and use `\__hook_label_ordered:nnTF` to apply the correct rule to the pair of labels, depending if it was sorted or not.

```
\cs_new_protected:Npn \__hook_rule_before_gset:nnn #1#2#3
  {
    \tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
      { \__hook_label_ordered:nnTF {#2} {#3} { < } { > } }
  }
\cs_new_eq:cN { __hook_rule_<_gset:nnn } \__hook_rule_before_gset:nnn
\cs_new_protected:Npn \__hook_rule_after_gset:nnn #1#2#3
  {
    \tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#3} {#2} _tl }
      { \__hook_label_ordered:nnTF {#3} {#2} { < } { > } }
  }
\cs_new_eq:cN { __hook_rule_>_gset:nnn } \__hook_rule_after_gset:nnn
```

(*End definition for* `\__hook_rule_before_gset:nnn` *and others.*)

`\__hook_rule_removes_gset:nnn` This rule removes (clears, actually) the code from label `#3` if label `#2` is in the hook `#1`.

```
\cs_new_protected:Npn \__hook_rule_removes_gset:nnn #1#2#3
  {
    \tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
      { \__hook_label_ordered:nnTF {#2} {#3} { -> } { <- } }
  }
```

(*End definition for* `\__hook_rule_removes_gset:nnn`.)

`\__hook_rule_incompatible-error_gset:nnn` `\__hook_rule_incompatible-warning_gset:nnn` These relations make an error/warning if labels `#2` and `#3` appear together in hook `#1`.

```
\cs_new_protected:cpn { __hook_rule_incompatible-error_gset:nnn } #1#2#3
  { \tl_gset:cn { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl } { xE } }
\cs_new_protected:cpn { __hook_rule_incompatible-warning_gset:nnn } #1#2#3
  { \tl_gset:cn { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl } { xW } }
```

(*End definition for* `\__hook_rule_incompatible-error_gset:nnn` *and* `\__hook_rule_incompatible-warning_gset:nnn`.)

`\__hook_rule_unrelated_gset:nnn` `\__hook_rule_gclear:nnn` Undo a setting. `\__hook_rule_unrelated_gset:nnn` doesn't need to do anything, since we use `\__hook_rule_gclear:nnn` before setting any rule.

```
\cs_new_protected:Npn \__hook_rule_unrelated_gset:nnn #1#2#3 { }
\cs_new_protected:Npn \__hook_rule_gclear:nnn #1#2#3
  { \cs_undefine:c { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl } }
```

(*End definition for* `\__hook_rule_unrelated_gset:nnn` *and* `\__hook_rule_gclear:nnn`.)

`\__hook_label_pair:nn` Ensure that the lexically greater label comes first.

```
\cs_new:Npn \__hook_label_pair:nn #1#2
  {
    \if_case:w \__hook_str_compare:nn {#1} {#2} \exp_stop_f:
          #1 | #1 %  0
```

```
    \or:   #1 | #2 % +1
    \else: #2 | #1 % -1
    \fi:
  }
```

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

`\__hook_label_ordered_p:nn` `\__hook_label_ordered:nnTF` Check that labels `#1` and `#2` are in the correct order (as returned by `\__hook_label_pair:nn`) and if so return true, else return false.

```
\prg_new_conditional:Npnn \__hook_label_ordered:nn #1#2 { TF }
  {
    \if_int_compare:w \__hook_str_compare:nn {#1} {#2} > 0 \exp_stop_f:
      \prg_return_true:
    \else
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\__hook_label_ordered:nnTF`.)

`\__hook_if_label_case:nnnnn` To avoid doing the string comparison twice in `\__hook_initialize_single:NNNNn` (once with `\str_if_eq:nn` and again with `\__hook_label_ordered:nn`), we use a three-way branching macro that will compare `#1` and `#2` and expand to `\use_i:nnn` if they are equal, `\use_ii:nn` if `#1` is lexically greater, and `\use_iii:nn` otherwise.

```
\cs_new:Npn \__hook_if_label_case:nnnnn #1#2
  {
    \cs:w use_
      \if_case:w \__hook_str_compare:nn {#1} {#2}
         i \or: ii \else: iii \fi: :nnn
    \cs_end:
  }
```

(*End definition for* `\__hook_if_label_case:nnnnn`.)

`\__hook_initialize_all:` Initialize all known hooks (at `\begin{document}`), i.e., update the fast execution token lists to hold the necessary code in the right order.

```
\cs_new_protected:Npn \__hook_initialize_all: {
```

First we change `\__hook_update_hook_code:n` which so far was a no-op to now initialize one hook. This way any later updates to the hook will run that code and also update the execution token list.

```
  \cs_gset_eq:NN \__hook_update_hook_code:n \__hook_initialize_hook_code:n
```

Now we loop over all hooks that have been defined and update each of them.

```
  \__hook_debug:n { \prop_gclear:N \g__hook_used_prop }
  \seq_map_inline:Nn \g__hook_all_seq
      {
        \__hook_update_hook_code:n {##1}
      }
```

If we are debugging we show results hook by hook for all hooks that have data.

```
  \__hook_debug:n
     { \iow_term:x{^^JAll~ initialized~ (non-empty)~ hooks:}
       \prop_map_inline:Nn \g__hook_used_prop
           { \iow_term:x{^^J~ ##1~ ->~
```

```
                \exp_not:v {g__hook_##1_code_tl}~ }
            }
        }
%
```

After all hooks are initialized we change the "use" to just call the hook code and not initialize it (as it was done in the preamble.

```
  \cs_gset_eq:NN \hook_use:n \__hook_use_initialized:n
  \cs_gset_eq:NN \__hook_preamble_hook:n \use_none:n
}
```

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

`\__hook_initialize_hook_code:n` Initializing or reinitializing the fast execution hook code. In the preamble this is selectively done in case a hook gets used and at `\begin{document}` this is done for all hooks and afterwards only if the hook code changes.

```
\cs_new_protected:Npn \__hook_initialize_hook_code:n #1 {
  \__hook_debug:n{ \iow_term:x{^^JUpdate~ code~ for~ hook~
                                  '#1' \on@line :^^J} }
```

This does the sorting and the updates. If there aren't any code chunks for the current hook, there is no point in even starting the sorting routine so we make a quick test for that and in that case just update `\g__hook_`⟨*hook*⟩`_code_tl` to hold the next code. If there are code chunks we call `\__hook_initialize_single:NNNNn` and pass to it ready made csnames as they are needed several times inside. This way we save a bit on processing time if we do that up front.

```
  \hook_if_exist:nT {#1}
    {
      \prop_if_empty:cTF {g__hook_#1_code_prop}
        { \tl_gset:co {g__hook_#1_code_tl}
                      {\cs:w g__hook_#1_next_code_tl \cs_end: } }
        {
```

By default the algorithm sorts the code chunks and then saves the result in a token list for fast execution by adding the code one after another using `\tl_gput_right:NV`. When we sort code for a reversed hook, all we have to do is to add the code chunks in the opposite order into the token list. So all we have to do in preparation is to change two definitions used later on.

```
          \__hook_if_reversed:nTF {#1}
            { \cs_set_eq:NN \__hook_tl_gput:NV    \tl_gput_left:NV
              \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_left:NV  }
            { \cs_set_eq:NN \__hook_tl_gput:NV    \tl_gput_right:NV
              \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_right:NV }
```

When sorting, some relations (namely `->` `<-`) need to act destructively on the code property lists to remove code that shouldn't appear in the sorted hook token list.

```
          \prop_gset_eq:Nc \g__hook_code_temp_prop { g__hook_#1_code_prop }
          \__hook_initialize_single:ccccn
            { g__hook_#1_code_prop } { g__hook_#1_code_tl }
            { g__hook_#1_next_code_tl } { g__hook_#1_labels_clist }
            {#1}
          \prop_gset_eq:cN { g__hook_#1_code_prop } \g__hook_code_temp_prop
```

For debug display we want to keep track of those hooks that actually got code added to them, so we record that in plist. We use a plist to ensure that we record each hook name only once, i.e., we are only interested in storing the keys and the value is arbitrary

```
            \__hook_debug:n{ \exp_args:NNx \prop_gput:Nnn \g__hook_used_prop {#1}{} }
          }
      }
}
```

(*End definition for* `\__hook_initialize_hook_code:n`.)

`\g__hook_used_prop` All hooks that receive code (for use in debugging display).

```
\prop_new:N\g__hook_used_prop
```

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

`\__hook_tl_csname:n` `\__hook_seq_csname:n` It is faster to pass a single token and expand it when necessary than to pass a bunch of character tokens around.

FMi: note to myself: verify

```
\cs_new:Npn \__hook_tl_csname:n #1 { l__hook_label_#1_tl }
\cs_new:Npn \__hook_seq_csname:n #1 { l__hook_label_#1_seq }
```

(*End definition for* `\__hook_tl_csname:n` *and* `\__hook_seq_csname:n`.)

`\l__hook_labels_seq` `\l__hook_labels_int` `\l__hook_front_tl` `\l__hook_rear_tl` `\l__hook_label_0_tl` For the sorting I am basically implementing Knuth's algorithm for topological sorting as given in TAOCP volume 1 pages 263–266. For this algorithm we need a number of local variables:

- List of labels used in the current hook to label code chunks:

```
    \seq_new:N \l__hook_labels_seq
```

- Number of labels used in the current hook. In Knuth's algorithm this is called N:

```
    \int_new:N \l__hook_labels_int
```

- The sorted code list to be build is managed using two pointers one to the front of the queue and one to the rear. We model this using token list pointers. Knuth calls them F and R:

```
    \tl_new:N \l__hook_front_tl
    \tl_new:N \l__hook_rear_tl
```

- The data for the start of the queue is kept in this token list, it corresponds to what Don calls `QLINK[0]` but since we aren't manipulating individual words in memory it is slightly differently done:

```
    \tl_new:c { \__hook_tl_csname:n { 0 } }
```

(*End definition for* `\l__hook_labels_seq` *and others.*)

__hook_initialize_single:NNNNn
__hook_initialize_single:ccccn

`\__hook_initialize_single:NNNNn` implements the sorting of the code chunks for a hook and saves the result in the token list for fast execution (`#3`). The arguments are ⟨*hook-code-plist*⟩, ⟨*hook-code-tl*⟩, ⟨*hook-next-code-tl*⟩, ⟨*hook-ordered-labels-clist*⟩ and ⟨*hook-name*⟩ (the latter is only used for debugging—the ⟨*hook-rule-plist*⟩ is accessed using the ⟨*hook-name*⟩).

The additional complexity compared to Don's algorithm is that we do not use simple positive integers but have arbitrary alphanumeric labels. As usual Don's data structures are chosen in a way that one can omit a lot of tests and I have mimicked that as far as possible. The result is a restriction I do not test for at the moment: a label can't be equal to the number 0!

FMi: Needs checking for, just in case

```
\cs_new_protected:Npn \__hook_initialize_single:NNNNn #1#2#3#4#5 {
  \debug_suspend:
```

Step T1: Initialize the data structure ...

```
  \seq_clear:N \l__hook_labels_seq
  \int_zero:N  \l__hook_labels_int
```

Store the name of the hook:

```
  \tl_set:Nn \l__hook_cur_hook_tl {#5}
```

We loop over the property list holding the code and record all labels listed there. Only rules for those labels are of interest to us. While we are at it we count them (which gives us the N in Knuth's algorithm. The prefix `label_` is added to the variables to ensure that labels named `front`, `rear`, `labels`, or `return` don't interact with our code.

```
  \prop_map_inline:Nn #1
    {
      \int_incr:N \l__hook_labels_int
      \seq_put_right:Nn \l__hook_labels_seq {##1}
      \tl_set:cn { \__hook_tl_csname:n {##1} }{0}     % the counter k for number of
                                                      % j before k rules
      \seq_clear_new:c { \__hook_seq_csname:n {##1} } % sequence of successors to k
                                                      % i.e., k before j rules (stores
                                                      % the names of the j's)
    }
```

Steps T2 and T3: Sort the relevant rules into the data structure...

This loop constitutes a square matrix of the labels in `#1` in the vertical and the horizontal directions. However since the rule $l_A\langle rel\rangle l_B$ is the same as $l_B\langle rel\rangle^{-1} l_A$ we can cut the loop short at the diagonal of the matrix (*i.e.*, when both labels are equal), saving a good amount of time. The way the rules were set up (see the implementation of `\__hook_rule_before_gset:nnn` above) ensures that we have no rule in the ignored side of the matrix, and all rules are seen. The rules are applied in `\__hook_apply_label_pair:nnn`, which takes the properly-ordered pair of labels as argument.

```
  \prop_map_inline:Nn #1
    {
      \prop_map_inline:Nn #1
        {
          \__hook_if_label_case:nnnnn {##1} {####1}
            { \prop_map_break: }
            { \__hook_apply_label_pair:nnn {##1} {####1} }
            { \__hook_apply_label_pair:nnn {####1} {##1} }
```

```
                {#5}
        }
    }
```

Take a breath and take a look at the data structures that have been set up:

```
  \__hook_debug:n { \__hook_debug_label_data:N #1 }
```

Step T4:

```
  \tl_set:Nn \l__hook_rear_tl { 0 }
  \tl_set:cn { \__hook_tl_csname:n { 0 } } { 0 } % really {l__hook_label_ \l__hook_rear_tl _
  \seq_map_inline:Nn \l__hook_labels_seq
      {
        \int_compare:nNnT { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
            {
              \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } }{##1}
              \tl_set:Nn \l__hook_rear_tl {##1}
            }
      }
  \tl_set_eq:Nc \l__hook_front_tl { \__hook_tl_csname:n { 0 } }
  \tl_gclear:N #2
  \clist_gclear:N #4
```

The whole loop combines steps T5–T7:

```
  \bool_while_do:nn { ! \str_if_eq_p:Vn \l__hook_front_tl { 0 } }
       {
```

This part is step T5:

```
          \int_decr:N \l__hook_labels_int
          \prop_get:NVN #1 \l__hook_front_tl \l__hook_return_tl
          \__hook_tl_gput:NV #2 \l__hook_return_tl
          \__hook_clist_gput:NV #4 \l__hook_front_tl
          \__hook_debug:n{ \iow_term:x{Handled~ code~ for~ \l__hook_front_tl} }
```

This is step T6 except that we don't use a pointer P to move through the successors, but instead use `##1` of the mapping function.

```
          \seq_map_inline:cn { \__hook_seq_csname:n { \l__hook_front_tl } }
              {
                \tl_set:cx { \__hook_tl_csname:n {##1} }
                    { \int_eval:n { \cs:w \__hook_tl_csname:n {##1} \cs_end: - 1 } }
                \int_compare:nNnT { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
                    {
                      \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } } {##1}
                      \tl_set:Nn \l__hook_rear_tl            {##1}
                    }
              }
```

and step T7:

```
          \tl_set_eq:Nc \l__hook_front_tl { \__hook_tl_csname:n { \l__hook_front_tl } }
```

This is step T8: If we haven't moved the code for all labels (i.e., if `\l__hook_labels_int` is still greater than zero) we have a loop and our partial order can't be flattened out.

```
       }
  \int_compare:nNnF \l__hook_labels_int = 0
      {
```

```
        \iow_term:x{====================}
        \iow_term:x{Error:~ label~ rules~ are~ incompatible:}
```

This is not really the information one needs in the error case but will do for now ...

FMi: fix

```
        \__hook_debug_label_data:N #1
        \iow_term:x{====================}
      }
```

After we have added all hook code to `#2` we finish it off with adding extra code for a one time execution. That is stored in `#3` but is normally empty.

```
  \tl_gput_right:Nn #2 {#3}
  \debug_resume:
}
\cs_generate_variant:Nn \__hook_initialize_single:NNNNn {cccc}
```

(*End definition for* `\__hook_initialize_single:NNNNn`.)

`\__hook_tl_gput:NV`
`\__hook_clist_gput:NV`

These append either on the right (normal hook) or on the left (reversed hook). This is setup up in `\__hook_initialize_hook_code:n`, elsewhere their behavior is undefined.

```
\cs_new:Npn \__hook_tl_gput:NV     {\ERROR}
\cs_new:Npn \__hook_clist_gput:NV  {\ERROR}
```

(*End definition for* `\__hook_tl_gput:NV` *and* `\__hook_clist_gput:NV`.)

`\__hook_apply_label_pair:nnn`
`\__hook_label_if_exist_apply:nnnF`

This is the payload of steps T2 and T3 executed in the loop described above. This macro assumes `#1` and `#2` are ordered, which means that any rule pertaining the pair `#1` and `#2` is `\g__hook_`⟨*hook*⟩`_rule_#1|#2_tl`, and not `\g__hook_`⟨*hook*⟩`_rule_#2|#1_tl`. This also saves a great deal of time since we only need to check the order of the labels once.

The arguments here are ⟨*label1*⟩, ⟨*label2*⟩, ⟨*hook*⟩, and ⟨*hook-code-plist*⟩. We are about to apply the next rule and enter it into the data structure. `\__hook_apply_label_pair:nnn` will just call `\__hook_label_if_exist_apply:nnnF` for the ⟨*hook*⟩, and if no rule is found, also try the ⟨*hook*⟩ name `??` denoting a default hook rule.

`\__hook_label_if_exist_apply:nnnF` will check if the rule exists for the given hook, and if so call `\__hook_apply_rule:nnn`.

```
\cs_new_protected:Npn \__hook_apply_label_pair:nnn #1#2#3
  {
```

Extra complication: as we use default rules and local hook specific rules we first have to check if there is a local rule and if that exist use it. Otherwise check if there is a default rule and use that.

```
    \__hook_label_if_exist_apply:nnnF {#1} {#2} {#3}
      {
```

If there is no hook-specific rule we check for a default one and use that if it exists.

```
        \__hook_label_if_exist_apply:nnnF {#1} {#2} { ?? } { }
      }
  }
\cs_new_protected:Npn \__hook_label_if_exist_apply:nnnF #1#2#3
  {
    \if_cs_exist:w g__hook_ #3 _rule_ #1 | #2 _tl \cs_end:
```

What to do precisely depends on the type of rule we have encountered. If it is a `before` rule it will be handled by the algorithm but other types need to be managed differently. All this is done in `\__hook_apply_rule:nnnN`.

```
      \__hook_apply_rule:nnn {#1} {#2} {#3}
      \exp_after:wN \use_none:n
    \else:
      \use:nn
    \fi:
  }
```

(*End definition for* `\__hook_apply_label_pair:nnn` *and* `\__hook_label_if_exist_apply:nnnF`.)

`\__hook_apply_rule:nnn` This is the code executed in steps T2 and T3 while looping through the matrix This is part of step T3. We are about to apply the next rule and enter it into the data structure. The arguments are ⟨*label1*⟩, ⟨*label2*⟩, ⟨*hook-name*⟩, and ⟨*hook-code-plist*⟩.

```
\cs_new_protected:Npn \__hook_apply_rule:nnn #1#2#3
  {
    \cs:w __hook_apply_
      \cs:w g__hook_#3_reversed_tl \cs_end: rule_
        \cs:w g__hook_ #3 _rule_ #1 | #2 _tl \cs_end: :nnn \cs_end:
      {#1} {#2} {#3}
  }
```

(*End definition for* `\__hook_apply_rule:nnn`.)

`\__hook_apply_rule_<:nnn` `\__hook_apply_rule_>:nnn` The most common cases are `<` and `>` so we handle that first. They are relations $\prec$ and $\succ$ in TAOCP, and they dictate sorting.

```
\cs_new_protected:cpn { __hook_apply_rule_<:nnn } #1#2#3
  {
    \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
    \tl_set:cx { \__hook_tl_csname:n {#2} }
       { \int_eval:n{ \cs:w \__hook_tl_csname:n {#2} \cs_end: + 1 } }
    \seq_put_right:cn{ \__hook_seq_csname:n {#1} }{#2}
  }
\cs_new_protected:cpn { __hook_apply_rule_>:nnn } #1#2#3
  {
    \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
    \tl_set:cx { \__hook_tl_csname:n {#1} }
       { \int_eval:n{ \cs:w \__hook_tl_csname:n {#1} \cs_end: + 1 } }
    \seq_put_right:cn{ \__hook_seq_csname:n {#2} }{#1}
  }
```

(*End definition for* `\__hook_apply_rule_<:nnn` *and* `\__hook_apply_rule_>:nnn`.)

`\__hook_apply_rule_xE:nnn` `\__hook_apply_rule_xW:nnn` These relations make two labels incompatible within a hook. `xE` makes raises an error if the labels are found in the same hook, and `xW` makes it a warning.

```
\cs_new_protected:cpn { __hook_apply_rule_xE:nnn } #1#2#3
  {
    \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
    \msg_error:nnnnnn { hooks } { labels-incompatible }
      {#1} {#2} {#3} { 1 }
    \use:c { __hook_apply_rule_->:nnn } {#1} {#2} {#3}
    \use:c { __hook_apply_rule_<-:nnn } {#1} {#2} {#3}
  }
```

```
\cs_new_protected:cpn { __hook_apply_rule_xW:nnn } #1#2#3
  {
    \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
    \msg_warning:nnnnnn { hooks } { labels-incompatible }
      {#1} {#2} {#3} { 0 }
  }
```

(*End definition for* `\__hook_apply_rule_xE:nnn` *and* `\__hook_apply_rule_xW:nnn`.)

`\__hook_apply_rule_->:nnn`
`\__hook_apply_rule_<-:nnn`

If we see `->` we have to drop code for label `#3` and carry on. We could do a little better and trop everything for that label since it doesn't matter where we sort in the empty code. However that would complicate the algorithm a lot with little gain. So we still unnecessarily try to sort it in and depending on the rules that might result in a loop that is otherwise resolved. If that turns out to be a real issue, we can improve the code.

Here the code is removed from `\l__hook_cur_hook_tl` rather than `#3` because the latter may be `??`, and the default hook doesn't store any code. Removing from `\l__hook_cur_hook_tl` makes default rules `->` and `<-` work properly.

```
\cs_new_protected:cpn { __hook_apply_rule_->:nnn } #1#2#3
  {
    \__hook_debug:n
      {
        \__hook_msg_pair_found:nnn {#1} {#2} {#3}
        \iow_term:x{--->~ Drop~ '#2'~ code~ from~
          \iow_char:N \\ g__hook_ \l__hook_cur_hook_tl _code_prop ~ because~ of~ '#1' }
      }
    \prop_gput:cnn { g__hook_ \l__hook_cur_hook_tl _code_prop } {#2} { }
  }
\cs_new_protected:cpn { __hook_apply_rule_<-:nnn } #1#2#3
  {
    \__hook_debug:n
      {
        \__hook_msg_pair_found:nnn {#1} {#2} {#3}
        \iow_term:x{--->~ Drop~ '#1'~ code~ from~
          \iow_char:N \\ g__hook_ \l__hook_cur_hook_tl _code_prop ~ because~ of~ '#2' }
      }
    \prop_gput:cnn { g__hook_ \l__hook_cur_hook_tl _code_prop } {#1} { }
  }
```

(*End definition for* `\__hook_apply_rule_->:nnn` *and* `\__hook_apply_rule_<-:nnn`.)

`\__hook_apply_-rule_<:nnn`
`\__hook_apply_-rule_>:nnn`
`\__hook_apply_-rule_<-:nnn`
`\__hook_apply_-rule_->:nnn`
`\__hook_apply_-rule_x:nnn`

Reversed rules.

```
\cs_new_eq:cc { __hook_apply_-rule_<:nnn  } { __hook_apply_rule_>:nnn }
\cs_new_eq:cc { __hook_apply_-rule_>:nnn  } { __hook_apply_rule_<:nnn }
\cs_new_eq:cc { __hook_apply_-rule_<-:nnn } { __hook_apply_rule_<-:nnn }
\cs_new_eq:cc { __hook_apply_-rule_->:nnn } { __hook_apply_rule_->:nnn }
\cs_new_eq:cc { __hook_apply_-rule_xE:nnn  } { __hook_apply_rule_xE:nnn }
\cs_new_eq:cc { __hook_apply_-rule_xW:nnn  } { __hook_apply_rule_xW:nnn }
```

(*End definition for* `\__hook_apply_-rule_<:nnn` *and others.*)

`\__hook_msg_pair_found:nnn`

A macro to avoid moving this many tokens around.

```
\cs_new_protected:Npn \__hook_msg_pair_found:nnn #1#2#3
  {
    \iow_term:x{~ \str_if_eq:nnTF {#3} {??} {default} {~normal} ~
```

```
                rule~ \__hook_label_pair:nn {#1} {#2}:~
              \use:c { g__hook_#3_rule_ \__hook_label_pair:nn {#1} {#2} _tl } ~ found}
  }
```

(*End definition for* `\__hook_msg_pair_found:nnn`.)

`\__hook_debug_label_data:N`

```
\cs_new_protected:Npn \__hook_debug_label_data:N #1 {
  \iow_term:x{Code~ labels~ for~ sorting:}
  \iow_term:x{~ \seq_use:Nnnn\l__hook_labels_seq {~and~}{,~}{~and~} }  % fix name!
  \iow_term:x{^^J Data~ structure~ for~ label~ rules:}
  \prop_map_inline:Nn #1
       {
         \iow_term:x{~ ##1~ =~ \tl_use:c{ \__hook_tl_csname:n {##1} }~ ->~
           \seq_use:cnnn{ \__hook_seq_csname:n {##1} }{~->~}{~->~}{~->~}
         }
       }
  \iow_term:x{}
}
```

(*End definition for* `\__hook_debug_label_data:N`.)

`\hook_log:n` This writes out information about the hook given in its argument onto the terminal and the `.log` file.

```
\cs_new_protected:Npn \hook_log:n #1
  {
    \exp_args:Nx \__hook_log:n
      { \__hook_parse_label_default:nn {#1} { top-level } }
  }
\cs_new_protected:Npn \__hook_log:n #1
  {
    \iow_term:x{^^JThe~ hook~ '#1':}
    \hook_if_exist:nF {#1}
      { \iow_term:x {~Hook~ is~ not~ declared!} }
    \__hook_if_exist:nTF {#1}
      {
        \iow_term:x{~Code~ chunks:}
        \prop_if_empty:cTF {g__hook_#1_code_prop}
          { \iow_term:x{\@spaces ---} }
          {
            \prop_map_inline:cn {g__hook_#1_code_prop}
              { \iow_term:x{\@spaces ##1~ ->~ \tl_to_str:n{##2} } }
          }
        \iow_term:x{~Extra~ code~ next~ invocation:}
        \iow_term:x{\@spaces
          \tl_if_empty:cTF { g__hook_#1_next_code_tl }
            {---} {->~ \str_use:c{g__hook_#1_next_code_tl} } }
```

FMi: This is currently only displaying the local rules, but it should also show the matching global rules!

```
        \iow_term:x{~Rules:}
        \prop_if_empty:cTF {g__hook_#1_rules_prop}
          { \iow_term:x{\@spaces ---} }
          { \prop_map_inline:cn {g__hook_#1_rules_prop}
              { \iow_term:x{\@spaces ##1~ with~ relation~ ##2} }
          }
        \hook_if_exist:nT {#1}
          { \iow_term:x { ~Execution~ order
               \prop_if_empty:cTF {g__hook_#1_rules_prop}
                 { \__hook_if_reversed:nT {#1}
                       { ~ (after~ reversal) }
                 }
                 { ~ (after~
                   \__hook_if_reversed:nT {#1} {reversal~ and~}
                   applying~ rules)
                 }
               :
              }
            \iow_term:x { \@spaces
              \clist_if_empty:cTF{g__hook_#1_labels_clist}
                 {not~ set~ yet}
                 { \clist_use:cnnn {g__hook_#1_labels_clist}
                                   { ,~ } { ,~ } { ,~ }   }
            }
          }
      }
      { \iow_term:n { ~The~hook~is~empty. } }
    \iow_term:n { }
  }
```

(*End definition for* `\hook_log:n`. *This function is documented on page* ??.)

3.7 Specifying code for next invocation

`\hook_gput_next_code:nn`

```
\cs_new_protected:Npn \hook_gput_next_code:nn #1
  {
    \exp_args:Nx \__hook_gput_next_code:nn
      { \__hook_parse_label_default:nn {#1} { top-level } }
  }
\cs_new_protected:Npn \__hook_gput_next_code:nn #1 #2
  {
    \__hook_declare:n {#1}
    \hook_if_exist:nTF {#1}
      { \__hook_gput_next_do:nn {#1} {#2} }
      { \__hook_try_declaring_generic_next_hook:nn {#1} {#2} }
  }
\cs_new_protected:Npn \__hook_gput_next_do:nn #1 #2
  {
    \tl_gput_right:cn { g__hook_#1_next_code_tl }
      { #2 \tl_gclear:c { g__hook_#1_next_code_tl } }
  }
```

(*End definition for* `\hook_gput_next_code:nn`. *This function is documented on page* 10.)

3.8 Using the hook

\hook_use:n
__hook_use_initialized:n
__hook_preamble_hook:n

`\hook_use:n` as defined here is used in the preamble, where hooks aren't initialized by default. `\__hook_use_initialized:n` is also defined, which is the non-`\protected` version for use within the document. Their definition is identical, except for the `\__hook_preamble_hook:n` (which wouldn't hurt in the expandable version, but it would be an unnecessary extra expansion).

`\__hook_use_initialized:n` holds the expandable definition while in the preamble. `\__hook_preamble_hook:n` initializes the hook in the preamble, and is redefined to `\use_none:n` at `\begin{document}`.

Both versions do the same internally: check if the hook exist as given, and if so use it as quickly as possible. If it doesn't exist, the a call to `\__hook_use:wn` checks for file hooks.

At `\begin{document}`, all hooks are initialized, and any change in them causes an update, so `\hook_use:n` can be made expandable. This one is better not protected so that it can expand into nothing if containing no code. Also important in case of generic hooks that we do not generate a `\relax` as a side effect of checking for a csname. In contrast to the TEX low-level `\csname ...\endcsname` construct `\tl_if_exist:c` is careful to avoid this.

```
\cs_new_protected:Npn \hook_use:n #1
  {
    \tl_if_exist:cTF { g__hook_#1_code_tl }
      {
        \__hook_preamble_hook:n {#1}
        \cs:w g__hook_#1_code_tl \cs_end:
      }
      { \__hook_use:wn #1 / \s__hook_mark {#1} }
  }
\cs_new:Npn \__hook_use_initialized:n #1
  {
    \tl_if_exist:cTF { g__hook_#1_code_tl }
      { \cs:w g__hook_#1_code_tl \cs_end: }
      { \__hook_use:wn #1 / \s__hook_mark {#1} }
  }
\cs_new_protected:Npn \__hook_preamble_hook:n #1
  { \__hook_initialize_hook_code:n {#1} }
```

(*End definition for* `\hook_use:n`, `\__hook_use_initialized:n`, *and* `\__hook_preamble_hook:n`. *This function is documented on page 9.*)

__hook_use:wn
__hook_try_file_hook:n
__hook_if_exist_use:n

`\__hook_use:wn` does a quick check to test if the current hook is a file hook: those need a special treatment. If it is not, the hook does not exist. If it is, then `\__hook_try_file_hook:n` is called, and checks that the current hook is a file-specific hook using `\__hook_if_file_hook:wTF`. If it's not, then it's a generic `file/` hook and is used if it exist.

If it is a file-specific hook, it passes through the same normalization as during declaration, and then it is used if defined. `\__hook_if_exist_use:n` checks if the hook exist, and calls `\__hook_preamble_hook:n` if so, then uses the hook.

```
\cs_new:Npn \__hook_use:wn #1 / #2 \s__hook_mark #3
  {
    \str_if_eq:nnTF {#1} { file }
      { \__hook_try_file_hook:n {#3} }
```

```
      { } % Hook doesn't exist
  }
\cs_new_protected:Npn \__hook_try_file_hook:n #1
  {
    \__hook_if_file_hook:wTF #1 / / \s__hook_mark
      {
        \exp_args:Ne \__hook_if_exist_use:n
          { \exp_args:Ne \__hook_file_hook_normalise:n {#1} }
      }
      { \__hook_if_exist_use:n {#1} } % file/ generic hook (e.g. file/before)
  }
\cs_new_protected:Npn \__hook_if_exist_use:n #1
  {
    \tl_if_exist:cT { g__hook_#1_code_tl }
      {
        \__hook_preamble_hook:n {#1}
        \cs:w g__hook_#1_code_tl \cs_end:
      }
  }
```

(*End definition for* `\__hook_use:wn`, `\__hook_try_file_hook:n`, *and* `\__hook_if_exist_use:n`.)

`\hook_use_once:n` For hooks that can and should be used only once we have a special use command that remembers the hook name in `\g__hook_execute_immediately_clist`. This has the effect that any further code added to the hook is executed immediately rather than stored in the hook.

```
\cs_new_protected:Npn \hook_use_once:n #1
  {
    \tl_if_exist:cT { g__hook_#1_code_tl }
      {
        \clist_gput_left:Nn \g__hook_execute_immediately_clist {#1}
        \hook_use:n {#1}
      }
  }
```

(*End definition for* `\hook_use_once:n`. *This function is documented on page* 9.)

3.9 Querying a hook

Simpler data types, like token lists, have three possible states; they can exist and be empty, exist and be non-empty, and they may not exist, in which case emptiness doesn't apply (though `\tl_if_empty:N` returns false in this case).

Hooks are a bit more complicated: they have four possible states. A hook may exist or not, and either way it may or may not be empty (even a hook that doesn't exist may be non-empty).

A hook is said to be empty when no code was added to it, either to its permanent code pool, or to its "next" token list. The hook doesn't need to be declared to have code added to its code pool (it may happen that a package *A* defines a hook `foo`, but it's loaded after package *B*, which adds some code to that hook. In this case it is important that the code added by package *B* is remembered until package *A* is loaded).

A hook is said to exist when it was declared with `\hook_new:n` or some variant thereof.

\hook_if_empty_p:n
\hook_if_empty:nTF

Test if a hook is empty (that is, no code was added to that hook). A hook being empty means that *both* its \g__hook_⟨*hook*⟩_code_prop and its \g__hook_⟨*hook*⟩_next_code_tl are empty.

```
\prg_new_conditional:Npnn \hook_if_empty:n #1 { p , T , F , TF }
  {
    \__hook_if_exist:nTF {#1}
      {
        \bool_lazy_and:nnTF
            { \prop_if_empty_p:c { g__hook_#1_code_prop } }
            { \tl_if_empty_p:c { g__hook_#1_next_code_tl } }
          { \prg_return_true: }
          { \prg_return_false: }
      }
      { \prg_return_true: }
  }
```

(*End definition for* \hook_if_empty:nTF. *This function is documented on page 10.*)

\hook_if_exist_p:n
\hook_if_exist:nTF

A canonical way to test if a hook exists. A hook exists if the token list that stores the sorted code for that hook, \g__hook_⟨*hook*⟩_code_tl, exists. The property list \g__hook_⟨*hook*⟩_code_prop cannot be used here because often it is necessary to add code to a hook without knowing if such hook was already declared, or even if it will ever be (for example, in case the package that defines it isn't loaded).

```
\prg_new_conditional:Npnn \hook_if_exist:n #1 { p , T , F , TF }
  {
    \tl_if_exist:cTF { g__hook_#1_code_tl }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* \hook_if_exist:nTF. *This function is documented on page 11.*)

__hook_if_exist_p:n
__hook_if_exist:nTF

An internal check if the hook has already been declared with __hook_declare:n. This means that the hook was already used somehow (a code chunk or rule was added to it), but it still wasn't declared with \hook_new:n.

```
\prg_new_conditional:Npnn \__hook_if_exist:n #1 { p , T , F , TF }
  {
    \prop_if_exist:cTF { g__hook_#1_code_prop }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* __hook_if_exist:nTF.)

__hook_if_reversed_p:n
__hook_if_reversed:nTF

An internal conditional that checks if a hook is reversed.

```
\prg_new_conditional:Npnn \__hook_if_reversed:n #1 { p , T , F , TF }
  {
    \if_int_compare:w \cs:w g__hook_#1_reversed_tl \cs_end: 1 < 0 \exp_stop_f:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\__hook_if_reversed:nTF`.)

`\g__hook_execute_immediately_clist` List of hooks that from no on should not longer receive code.

```
\clist_new:N \g__hook_execute_immediately_clist
```

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

3.10 Messages

```
\msg_new:nnnn { hooks } { labels-incompatible }
  {
    Labels~‘#1’~and~‘#2’~are~incompatible
    \str_if_eq:nnF {#3} {??} { ~in~hook~‘#3’ } .~
    \int_compare:nNnT {#4} = { 1 }
      { The~code~for~both~labels~will~be~dropped. }
  }
  {
    LaTeX~found~two~incompatible~labels~in~the~same~hook.~
    This~indicates~an~incompatibility~between~packages.
  }
\msg_new:nnn { hooks } { empty-label }
  { Empty~code~label~\msg_line_context:.~Using~‘#1’~instead. }
```

3.11 LaTeX 2_ε package interface commands

`\NewHook` `\NewReversedHook` `\NewMirroredHookPair` Declaring new hooks ...

```
\NewDocumentCommand \NewHook             { m }{ \hook_new:n {#1} }
\NewDocumentCommand \NewReversedHook     { m }{ \hook_new_reversed:n {#1} }
\NewDocumentCommand \NewMirroredHookPair { mm }{ \hook_new_pair:nn {#1}{#2} }
```

(*End definition for* `\NewHook`, `\NewReversedHook`, *and* `\NewMirroredHookPair`. *These functions are documented on page* 3.)

`\AddToHook`

```
\NewDocumentCommand \AddToHook { m o +m }
  {
    \clist_if_in:NnTF \g__hook_execute_immediately_clist {#1}
      {#3}
      { \hook_gput_code:nnn {#1} {#2} {#3} }
  }
```

(*End definition for* `\AddToHook`. *This function is documented on page* 4.)

`\AddToHookNext`

```
\NewDocumentCommand \AddToHookNext { m +m }
  { \hook_gput_next_code:nn {#1} {#2} }
```

(*End definition for* `\AddToHookNext`. *This function is documented on page* 5.)

`\RemoveFromHook`

```
\NewDocumentCommand \RemoveFromHook { m o }
  { \hook_gremove_code:nn {#1} {#2} }
```

(*End definition for* `\RemoveFromHook`. *This function is documented on page* 4.)

\DeclareDefaultHookLabel
__hook_curr_name_push:n
__hook_curr_name_pop:

The token list `\g__hook_hook_curr_name_tl` stores the name of the current package/file to be used as label for hooks. Providing a consistent interface is tricky, because packages can be loaded within packages, and some packages may not use `\DeclareDefaultHookLabel` to change the default label (in which case `\@currname` is used, if set).

To pull that off, we keep a stack that contains the default label for each level of input. The bottom of the stack contains the default label for the top-level. Since the string `top-level` is hardcoded, here this item of the stack is empty. Also, since we're in an input level, add `lthooks` to the stack as well. This stack should never go empty, so we loop through LaTeX 2_ε's file name stack, and add empty entries to `\g__hook_name_stack_seq` for each item in that stack. The last item is the `top-level`, which also gets an empty entry.

Also check for the case we're loading `lthooks` in the LaTeX 2_ε kernel. In that case, `\@currname` isn't `lthooks` and just the top-level is added to the stack as an empty entry.

```
\str_if_eq:VnTF \@currname { lthooks }
  {
    \seq_gpush:Nn \g__hook_name_stack_seq { lthooks }
    \cs_set_protected:Npn \__hook_tmp:w #1 #2 #3
      {
        \quark_if_recursion_tail_stop:n {#1}
        \seq_gput_right:Nn \g__hook_name_stack_seq { }
        \__hook_tmp:w
      }
    \exp_after:wN \__hook_tmp:w
      \@currnamestack
      \q_recursion_tail \q_recursion_tail
      \q_recursion_tail \q_recursion_stop
  }
  { \seq_gpush:Nn \g__hook_name_stack_seq { } }
```

Two commands keep track of the stack: when a file is input, `\__hook_curr_name_push:n` pushes an (empty by default) label to the stack:

```
\cs_new_protected:Npn \__hook_curr_name_push:n #1
  {
    \seq_gpush:Nn \g__hook_name_stack_seq {#1}
    \tl_gset:Nn \g__hook_hook_curr_name_tl {#1}
  }
%
```

and when an input is over, the topmost item of the stack is popped, since the label will not be used again, and `\g__hook_hook_curr_name_tl` is updated to the now topmost item of the stack:

```
\cs_new_protected:Npn \__hook_curr_name_pop:
  {
    \seq_gpop:NN \g__hook_name_stack_seq \l__hook_return_tl
    \seq_get:NNTF \g__hook_name_stack_seq \l__hook_return_tl
      { \tl_gset_eq:NN \g__hook_hook_curr_name_tl \l__hook_return_tl }
      { \ERROR_should_not_happen }
  }
```

The token list `\g__hook_hook_curr_name_tl` is but a mirror of the top of the stack.

Now define a wrapper that replaces the top of the stack with the argument, and updates `\g__hook_hook_curr_name_tl` accordingly.

```
\NewDocumentCommand \DeclareDefaultHookLabel { m }
  {
    \seq_gpop:NN \g__hook_name_stack_seq \l__hook_return_tl
    \__hook_curr_name_push:n {#1}
  }
%    \begin{macrocode}
%
%   The push and pop macros are injected in \cs{@pushfilename} and
%   \cs{@popfilename} so that they correctly keep track of the label.s
%    \begin{macrocode}
% TODO! \pho{Properly integrate in the kernel}
\tl_gput_left:Nn \@pushfilename { \__hook_curr_name_push:n { } }
\tl_gput_left:Nn \@popfilename { \__hook_curr_name_pop: }
% TODO! \pho{Properly integrate in the kernel}
```

(*End definition for* `\DeclareDefaultHookLabel`, `\__hook_curr_name_push:n`, *and* `\__hook_curr_name_pop:`. *This function is documented on page* 6.)

`\UseHook` Avoid the overhead of xparse and its protection that we don't want here (since the hook should vanish without trace if empty)!

```
\newcommand \UseHook { \hook_use:n }
```

(*End definition for* `\UseHook`. *This function is documented on page* 3.)

`\UseOneTimeHook`

```
\cs_new_protected:Npn \UseOneTimeHook { \hook_use_once:n }
```

(*End definition for* `\UseOneTimeHook`. *This function is documented on page* 3.)

`\ShowHook`

```
\cs_new_protected:Npn \ShowHook { \hook_log:n }
```

(*End definition for* `\ShowHook`. *This function is documented on page* 9.)

`\DebugHookOn`
`\DebugHookOff`

```
\cs_new_protected:Npn \DebugHookOn { \hook_debug_on: }
\cs_new_protected:Npn \DebugHookOff { \hook_debug_off: }
```

(*End definition for* `\DebugHookOn` *and* `\DebugHookOff`. *These functions are documented on page* 9.)

`\DeclareHookRule`

```
\NewDocumentCommand \DeclareHookRule { m m m m }
{ \hook_gset_rule:nnnn {#1}{#2}{#3}{#4} }
```

(*End definition for* `\DeclareHookRule`. *This function is documented on page* 7.)

`\DeclareDefaultHookRule` This declaration is only supported before `\begin{document}`.

```
\NewDocumentCommand \DeclareDefaultHookRule { m m m }
                    { \hook_gset_rule:nnnn {??}{#1}{#2}{#3} }
\@onlypreamble\DeclareDefaultHookRule
```

(*End definition for* `\DeclareDefaultHookRule`. *This function is documented on page* 7.)

`\ClearHookRule` A special setup rule that removes an existing relation. Basically @@_rule_gclear:nnn plus fixing the property list for debugging.

FMi: Need an L3 interface, or maybe it should get dropped

```
\NewDocumentCommand \ClearHookRule { m m m }
{ \hook_gset_rule:nnnn {#1}{#2}{unrelated}{#3} }
```

(*End definition for* `\ClearHookRule`*. This function is documented on page* 7.)

`\IfHookExistTF`
`\IfHookEmptyTF`

```
\NewExpandableDocumentCommand \IfHookExistTF { m }
  { \hook_if_exist:nTF {#1} }
\NewExpandableDocumentCommand \IfHookEmptyTF { m }
  { \hook_if_empty:nTF {#1} }
```

(*End definition for* `\IfHookExistTF` *and* `\IfHookEmptyTF`*. These functions are documented on page* 8.)

`\AtBeginDocument`

```
\renewcommand\AtBeginDocument{\AddToHook{begindocument}}
```

(*End definition for* `\AtBeginDocument`*. This function is documented on page* 14.)

`\AtEndDocument`

```
\renewcommand\AtEndDocument {\AddToHook{enddocument}}
%\renewcommand\AtEndDocument {\AddToHook{env/document/end}} % alternative impl
```

(*End definition for* `\AtEndDocument`*. This function is documented on page* 14.)

3.12 Set up existing LaTeX 2_ε hooks

As we are in a package calling `\NewHook` would label any already set up hook code under the package name, but we want it under the name `top-level` so we pretend that `\@currname` is empty.

```
\begingroup
  \def\@currname{}
  \NewHook{begindocument}
  \NewHook{enddocument}
```

We need to initialize the mechanism at `\begin{document}` but obviously before everything else, so we sneak[8] `\__hook_initialize_all:` into the LaTeX 2_ε hook name.

We can't use `\tl_gput_left:Nn` because that complains about `\@begindocumenthook` not starting with `\g_` so we do this through the backdoor.

```
%  \tex_global:D\tl_put_left:Nn \@begindocumenthook
%      {\__hook_initialize_all:}
```

There aren't many other hooks at the moment:

```
  \NewHook{rmfamily}
  \NewHook{sffamily}
  \NewHook{ttfamily}
  \NewHook{defaultfamily}
```

Not checked what this one does and whether it should be there (or is a real "hook".

```
  \NewHook{documentclass}
\endgroup
```

[8] This needs to move to `\document` directly.

4 Generic hooks for environments

```
\let\begin\relax  % avoid redeclaration message
\DeclareRobustCommand*\begin[1]{%
  \UseHook{env/#1/before}%
  \@ifundefined{#1}%
    {\def\reserved@a{\@latex@error{Environment~#1~undefined}\@eha}}%
    {\def\reserved@a{\def\@currenvir{#1}%
        \edef\@currenvline{\on@line}%
        \@execute@begin@hook{#1}%
        \csname #1\endcsname}}%
  \@ignorefalse
  \begingroup\@endpefalse\reserved@a}
```

Before the `\document` code is executed we have to first undo the `\endgroup` as there should be none for this environment to avoid that changes on top-level unnecessarily go to TEX's savestack, and we have to initialize all hooks in the hook system. So we need to test for this environment name. But once it has be found all this testing is no longer needed and so we redefine `\@execute@begin@hook` to simply use the hook

```
\def\@execute@begin@hook #1{%
  \expandafter\ifx\csname #1\endcsname\document
    \endgroup
    \gdef\@execute@begin@hook##1{\UseHook{env/##1/begin}}%
    \__hook_initialize_all:
    \@execute@begin@hook{#1}%
```

If this is an environment before `\begin{document}` we just run the hook.

```
  \else
    \UseHook{env/#1/begin}%
  \fi
}
\@namedef{end~}#1{%
  \UseHook{env/#1/end}%
  \csname end#1\endcsname\@checkend{#1}%
  \expandafter\endgroup\if@endpe\@doendpe\fi
  \UseHook{env/#1/after}%
  \if@ignore\@ignorefalse\ignorespaces\fi}%
```

Version that fixes tlb3722 but the change should perhaps be made in tabularx instead.

```
\@namedef{end~}#1{%
\romannumeral
\IfHookEmptyTF{env/#1/end}%
  {\expandafter\z@}%
  {\z@\UseHook{env/#1/end}}%
\csname end#1\endcsname\@checkend{#1}%
\expandafter\endgroup\if@endpe\@doendpe\fi
\UseHook{env/#1/after}%
\if@ignore\@ignorefalse\ignorespaces\fi}%
```

We provide 4 high-level hook interfaces directly, the others only when etoolbox is loaded

```
\newcommand\AtBeginEnvironment[1]    {\AddToHook{env/#1/begin}}
\newcommand\AtEndEnvironment[1]      {\AddToHook{env/#1/end}}
\newcommand\BeforeBeginEnvironment[1]{\AddToHook{env/#1/before}}
\newcommand\AfterEndEnvironment[1]   {\AddToHook{env/#1/after}}
```

5 Generic hooks for file loads

6 Hooks in \begin document

Can't have `@@` notation here as this is LaTeX 2_ε code ... and makes for puzzling errors if the double `@` signs get substituted.

```
⟨@@=⟩
\ExplSyntaxOff
```

The `begindocument` hook was already set up earlier, here is now the additional one (which was originally from the etoolbox package under the name `afterpreamble`.

```
\NewHook{begindocument/end}
```

`\document`

```
\def\document{%
```

We do cancel the grouping as part of the `\begin` handling (this is now done inside `\begin` instead) so that the `env/`⟨*env*⟩`/begin` hook is not hidden inside `\begingroup ... \endgroup`.

```
%  \endgroup
  \@kernel@after@env@document@begin
```

Added hook to load l3backend code:

```
  \@expl@sys@load@backend@@
  \ifx\@unusedoptionlist\@empty\else
    \@latex@warning@no@line{Unused global option(s):^^J%
             \@spaces[\@unusedoptionlist]}%
  \fi
  \@colht\textheight
  \@colroom\textheight \vsize\textheight
  \columnwidth\textwidth
  \@clubpenalty\clubpenalty
  \if@twocolumn
    \advance\columnwidth -\columnsep
    \divide\columnwidth\tw@ \hsize\columnwidth \@firstcolumntrue
  \fi
  \hsize\columnwidth \linewidth\hsize
  \begingroup\@floatplacement\@dblfloatplacement
    \makeatletter\let\@writefile\@gobbletwo
    \global \let \@multiplelabels \relax
    \@input{\jobname.aux}%
  \endgroup
  \if@filesw
    \immediate\openout\@mainaux\jobname.aux
    \immediate\write\@mainaux{\relax}%
  \fi
  \process@table
  \let\glb@currsize\@empty  % Force math initialization.
  \normalsize
  \everypar{}%
  \ifx\normalsfcodes\@empty
    \ifnum\sfcode`\.=\@m
      \let\normalsfcodes\frenchspacing
```

```
    \else
      \let\normalsfcodes\nonfrenchspacing
    \fi
  \fi
  \ifx\document@default@language\m@ne
    \chardef\document@default@language\language
  \fi
  \@noskipsecfalse
  \let \@refundefined \relax
%  \let\AtBeginDocument\@firstofone
%  \@begindocumenthook
  \UseOneTimeHook{begindocument}%
  \@kernel@after@begindocument
  \ifdim\topskip<1sp\global\topskip 1sp\relax\fi
  \global\@maxdepth\maxdepth
  \global\let\@begindocumenthook\@undefined
  \ifx\@listfiles\@undefined
    \global\let\@filelist\relax
    \global\let\@addtofilelist\@gobble
  \fi
  \gdef\do##1{\global\let ##1\@notprerr}%
  \@preamblecmds
  \global\let \@nodocument \relax
  \global\let\do\noexpand
  \UseOneTimeHook{begindocument/end}%
  \ignorespaces}
\let\@kernel@after@begindocument\@empty
```

(*End definition for* `\document`. *This function is documented on page* **??**.)

`\@kernel@after@env@document@begin`
`\@kernel@hook@begindocument`

```
\edef \@kernel@after@env@document@begin{%
  \let\expandafter\noexpand\csname
       g__hook_env/document/begin_code_tl\endcsname
  \noexpand\@empty}
\let\@kernel@hook@begindocument\@empty
```

(*End definition for* `\@kernel@after@env@document@begin` *and* `\@kernel@hook@begindocument`. *These functions are documented on page* **??**.)

7 Hooks in \enddocument

The `enddocument` hook was already set up earlier, here are now the additional ones:

```
\NewHook{enddocument/afterlastpage}
\NewHook{enddocument/afteraux}
\NewHook{enddocument/info}
\NewHook{enddocument/end}
```

`\enddocument`

```
\def\enddocument{%
   \UseHook{enddocument}%
```

```
    \@kernel@after@enddocument
    \@checkend{document}%
    \clearpage
    \UseHook{enddocument/afterlastpage}%
    \@kernel@after@enddocument@afterlastpage
    \begingroup
      \if@filesw
        \immediate\closeout\@mainaux
        \let\@setckpt\@gobbletwo
        \let\@newl@bel\@testdef
        \@tempswafalse
        \makeatletter \@@input\jobname.aux
      \fi
      \UseHook{enddocument/afteraux}%
```

Next hook is expect to contain only code for writing info messages on the terminal.

```
      \UseHook{enddocument/info}%
    \endgroup
    \UseHook{enddocument/end}%
    \deadcycles\z@\@@end}
```

The two kernel hooks above are used by the shipout code.

```
\let\@kernel@after@enddocument\@empty
\let\@kernel@after@enddocument@afterlastpage\@empty
```

(*End definition for* `\enddocument`. *This function is documented on page* **??**.)

`\@enddocument@kernel@warnings`

```
\def\@enddocument@kernel@warnings{%
    \ifdim \font@submax >\fontsubfuzz\relax
      \@font@warning{Size substitutions with differences\MessageBreak
                  up to \font@submax\space have occurred.\@gobbletwo}%
    \fi
    \@defaultsubs
    \@refundefined
    \if@filesw
      \ifx \@multiplelabels \relax
        \if@tempswa
          \@latex@warning@no@line{Label(s) may have changed.
              Rerun to get cross-references right}%
        \fi
      \else
        \@multiplelabels
      \fi
    \fi
}
\AddToHook{enddocument/info}[kernel/filelist]{\@dofilelist}
\AddToHook{enddocument/info}[kernel/warnings]{\@enddocument@kernel@warnings}
\DeclareHookRule{enddocument/info}{kernel/filelist}{before}{kernel/warnings}
```

(*End definition for* `\@enddocument@kernel@warnings`. *This function is documented on page* **??**.)

7.1 Adjusting at atveryend interfaces

With the new hook management all of atveryend is taken care of.

We therefore prevent the package from loading:

```
\expandafter\let\csname ver@atveryend.sty\endcsname\fmtversion
```

Here are new definitions for its interfaces now pointing to the hooks in `\enddocument`

```
\newcommand\AfterLastShipout  {\AddToHook{enddocument/afterlastpage}}
\newcommand\AtVeryEndDocument {\AddToHook{enddocument/afteraux}}
```

Next one is a bit of a fake, but the result should normally be as expected. If not one needs to add a rule to sort the code chunks in `enddocument/info`.

```
\newcommand\AtEndAfterFileList{\AddToHook{enddocument/info}}
\newcommand\AtVeryVeryEnd     {\AddToHook{enddocument/end}}
```

`\BeforeClearDocument` This one is the only one we don't implement or rather don't have a dedicated hook in the code.

```
\ExplSyntaxOn
\newcommand\BeforeClearDocument[1]
  { \AtEndDocument{#1}
    \@DEPRECATED{BeforeClearDocument \tl_to_str:n{#1}}
  }
\cs_new:Npn\@DEPRECATED #1
   {\iow_term:x{======~DEPRECATED~USAGE~#1~==========}}
\ExplSyntaxOff
```

(*End definition for* `\BeforeClearDocument`*. This function is documented on page* **??**.)

```
⟨/2ekernel⟩
```

8 A package version of the code for testing

```
⟨*package⟩
\RequirePackage{xparse}
\ProvidesExplPackage{lthooks}{\lthooksdate}{\lthooksversion}
                     {Hook management interface for LaTeX2e}
```

8.1 Core hook management code (kernel part)

This should run in older formats so we can't use `\IfFormatAtLeastTF` right now.

```
\@ifl@t@r\fmtversion{2020/10/01}
                     {}
                     {\input{lthooks.ltx}
                      \input{ltshipout.ltx}
                      \input{ltfilehook.ltx}
                     }
```

8.2 Package options

For now we offer a simple debug option which turns on a lot of strange `\typeout` messages, nothing fancy.

```
\ExplSyntaxOn

\hook_debug_off:
\DeclareOption { debug } { \hook_debug_on:
                           \shipout_debug_on: }
```

For now we offer a simple debug option which turns on a lot of strange `\typeout` messages, nothing fancy.

```
\shipout_debug_off:
\DeclareOption { debug-shipout } { \shipout_debug_on: }

\ProcessOptions
```

8.3 Temporarily patching package until changed

filehook support until that package is patched:

```
\RequirePackage{filehook-ltx}

⟨/package⟩
```

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.

D

E

F

G

H

I

S

T

U

V

W