LaTeX's hook management[*]

Frank Mittelbach[†]

December 6, 2021

Contents

---

[*]This module has version v1.0s dated 2021/09/28, © 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 2_ε 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 2ε interfaces

2.1.1 Declaring hooks

With a few exceptions, hooks have to be declared before they can be used. The exceptions are the generic hooks for commands and environments (executed at `\begin` and `\end`), and the hooks run when loading files (see section 3.1).

`\NewHook`

`\NewHook {⟨hook⟩}`

Creates a new ⟨hook⟩. If this hook is declared within a package it is suggested that its 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 2.1.5.

`\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 2.1.5.

`\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 2.1.5.

2.1.2 Special declarations for generic hooks

The declarations here should normally not be used. They are available to provide support for special use cases mainly involving generic command hooks.

`\DisableGenericHook`

`\DisableGenericHook {⟨hook⟩}`

After this declaration[1] the ⟨hook⟩ is no longer usable: Any attempt to add further code to it will result in an error and any use, e.g., via `\UseHook`, will simply do nothing.

This is intended to be used with generic command hooks (see `ltcmdhooks-doc`) as depending on the definition of the command such generic hooks may be unusable. If that is known, a package developer can disable such hooks up front.

The ⟨hook⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

`\ActivateGenericHook`

`\ActivateGenericHook {⟨hook⟩}`

This declaration activates a generic hook provided by a package/class (e.g., one used in code with `\UseHook` or `\UseOneTimeHook`) without it being explicitly declared with `\NewHook`). This command undoes the effect of `\DisableGenericHook`. If the hook is already activated, this command does nothing.

See section 2.6 for a discussion of when this declaration is appropriate.

---

[1]In the 2020/06 release this command was called `\DisableHook`, but that name was misleading as it shouldn't be used to disable non-generic hooks.

3

2.1.3 Using hooks in code

\UseHook

\UseHook {⟨*hook*⟩}

Execute the hook code inside a command or environment.

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.

The ⟨*hook*⟩ *cannot* be specified using the dot-syntax. A leading . is treated literally. See section 2.1.5 for details.

Using \UseOneTimeHook several times with the same {⟨*hook*⟩} means that it only executes the first time it is used. For example, if it is used in a command that can be called several times then the hook executes during only the *first* invocation of that command; this allows its use as an "initialization hook".

Mixing \UseHook and \UseOneTimeHook for the same {⟨*hook*⟩} should be avoided, but if this is done then neither will execute after the first \UseOneTimeHook.

2.1.4 Updating code for hooks

\AddToHook

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

Adds ⟨*code*⟩ to the ⟨*hook*⟩ labeled by ⟨*label*⟩. When the optional argument ⟨*label*⟩ is not provided, the ⟨*default label*⟩ is used (see section 2.1.5). If \AddToHook is used in a package/class, the ⟨*default label*⟩ is the package/class name, otherwise it is top-level (the top-level label is treated differently: see section 2.1.6).

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, 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 hooks from other packages without worrying whether they are actually used in the current document. See section 2.1.8.

The ⟨*hook*⟩ and ⟨*label*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

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

Removes any code labeled by ⟨*label*⟩ from the ⟨*hook*⟩. When the optional argument ⟨*label*⟩ is not provided, the ⟨*default label*⟩ is used (see section 2.1.5).

If there is no code under the ⟨*label*⟩ in the ⟨*hook*⟩, or if the ⟨*hook*⟩ does not exist, a warning is issued when you attempt to \RemoveFromHook, and the command is ignored. \RemoveFromHook should be used only when you know exactly what labels are in a hook. Typically this will be when some code gets added to a hook by a package, then later this code is removed by that same package. If you want to prevent the execution of code from another package, use the `voids` rule instead (see section 2.1.7).

If the optional ⟨*label*⟩ argument is *, then all code chunks are removed. This is rather dangerous as it may well drop code from other packages (that one may not know about); it should therefore not be used in packages but only in document preambles!

The ⟨*hook*⟩ and ⟨*label*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

In contrast to the `voids` relationship between two labels in a \DeclareHookRule this is a destructive operation as the labeled code is removed from the hook data structure, whereas the relationship setting can be undone by providing a different relationship later.

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 two 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 this 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 of the hook happens after the end of that 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]

If this declaration is used with a one-time hook then the code is only ever used if the declaration comes before the hook's invocation. This is because, in contrast to **\AddToHook**, the code in this declaration is not executed immediately in the case when the invocation of the hook has already happened—in other words, this code will truly execute only on the next invocation of the hook (and in the case of a one-time hook there is no such "next invocation"). This gives you a choice: should my code execute always, or should it execute only at the point where the one-time hook is used (and not at all if this is impossible)? For both of these possibilities there are use cases.

It is possible to nest this declaration using the same hook (or different hooks): e.g.,

\AddToHookNext{⟨*hook*⟩}{⟨*code-1*⟩\AddToHookNext{⟨*hook*⟩}{⟨*code-2*⟩}}

will execute ⟨*code-1*⟩ next time the ⟨*hook*⟩ is used and at that point puts ⟨*code-2*⟩ into the ⟨*hook*⟩ so that it gets executed on following time the hook is run.

A 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 2.1.8.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\ClearHookNext \ClearHookNext{⟨*hook*⟩}

Normally **\AddToHookNext** is only used when you know precisely where it will apply and why you want some extra code at that point. However, there are a few use cases in which such a declaration needs to be canceled, for example, when discarding a page with **\DiscardShipoutBox** (but even then not always), and in such situations **\ClearHookNext** can be used.

2.1.5 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.

Except for **\UseHook**, **\UseOneTimeHook** and **\IfHookEmptyTF** (and their expl3 interfaces **\hook_use:n**, **\hook_use_once:n** and **\hook_if_empty:nTF**), all ⟨*hook*⟩ and ⟨*label*⟩ arguments are processed in the same way: first, spaces are trimmed around the argument, then it is fully expanded until only character tokens remain. If the full expansion

---

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

of the ⟨*hook*⟩ or ⟨*label*⟩ contains a non-expandable non-character token, a low-level TEX error is raised (namely, the ⟨*hook*⟩ is expanded using TEX's `\csname`...`\endcsname`, as such, Unicode characters are allowed in ⟨*hook*⟩ and ⟨*label*⟩ arguments). The arguments of `\UseHook`, `\UseOneTimeHook`, and `\IfHookEmptyTF` are processed much in the same way except that spaces are not trimmed around the argument, for better performance.

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 the ⟨*hook*⟩ name or the ⟨*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 `\SetDefaultHookLabel`). 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}{.}{before}{babel}
\AddToHook {file/foo.tex/after}{code}
```

are equivalent to:

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

The ⟨*default label*⟩ is automatically set equal to the name of the current package or class at the time the package is loaded. If 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 may have exceptions—see `\PushDefaultHookLabel`.

This syntax is available in all ⟨*label*⟩ arguments and most ⟨*hook*⟩ arguments, both in the LATEX 2$_\varepsilon$ 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`), 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 `\SetDefaultHookLabel` or `\PushDefaultHookLabel{..}`...`\PopDefaultHookLabel`.

`\PushDefaultHookLabel`	`\PushDefaultHookLabel {⟨default label⟩}`
`\PopDefaultHookLabel`	⟨code⟩
	`\PopDefaultHookLabel`

`\PushDefaultHookLabel` sets the current ⟨*default label*⟩ to be used in ⟨*label*⟩ arguments, or when replacing a leading "." (see above). `\PopDefaultHookLabel` reverts the ⟨*default label*⟩ to its previous value.

Inside a package or class, the ⟨*default label*⟩ is equal to the package or class name, unless explicitly changed. Everywhere else, the ⟨*default label*⟩ is `top-level` (see section 2.1.6) unless explicitly changed.

The effect of `\PushDefaultHookLabel` holds until the next `\PopDefaultHookLabel`. `\usepackage` (and `\RequirePackage` and `\documentclass`) internally use

    `\PushDefaultHookLabel{`⟨*package name*⟩`}`
      ⟨*package code*⟩
    `\PopDefaultHookLabel`

to set the ⟨*default label*⟩ for the package or class file. Inside the ⟨*package code*⟩ the ⟨*default label*⟩ can also be changed with `\SetDefaultHookLabel`. `\input` and other file input-related commands from the LATEX kernel do not use `\PushDefaultHookLabel`, so code within files loaded by these commands does *not* get a dedicated ⟨*label*⟩! (that is, the ⟨*default label*⟩ is the current active one when the file was loaded.)

Packages that provide their own package-like interfaces (Ti*k*Z's `\usetikzlibrary`, for example) can use `\PushDefaultHookLabel` and `\PopDefaultHookLabel` to set dedicated labels and to emulate `\usepackage`-like hook behavior within those contexts.

The `top-level` label is treated differently, and is reserved to the user document, so it is not allowed to change the ⟨*default label*⟩ to `top-level`.

`\SetDefaultHookLabel`	`\SetDefaultHookLabel {⟨default label⟩}`

Similarly to `\PushDefaultHookLabel`, sets the current ⟨*default label*⟩ to be used in ⟨*label*⟩ arguments, or when replacing a leading ".". The effect holds until the label is changed again or until the next `\PopDefaultHookLabel`. The difference between `\PushDefaultHookLabel` and `\SetDefaultHookLabel` is that the latter does not save the current ⟨*default label*⟩.

This command is useful when a large package is composed of several smaller packages, but all should have the same ⟨*label*⟩, so `\SetDefaultHookLabel` can be used at the beginning of each package file to set the correct label.

`\SetDefaultHookLabel` is not allowed in the main document, where the ⟨*default label*⟩ is `top-level` and there is no `\PopDefaultHookLabel` to end its effect. It is also not allowed to change the ⟨*default label*⟩ to `top-level`.

2.1.6 The `top-level` label

The `top-level` label, assigned to code added from the main document, is different from other labels. Code added to hooks (usually `\AtBeginDocument`) in the preamble is almost always to change something defined by a package, so it should go at the very end of the hook.

Therefore, code added in the `top-level` is always executed at the end of the hook, regardless of where it was declared. If the hook is reversed (see `\NewReversedHook`), the `top-level` chunk is executed at the very beginning instead.

Rules regarding `top-level` have no effect: if a user wants to have a specific set of rules for a code chunk, they should use a different label to said code chunk, and provide a rule for that label instead.

The `top-level` label is exclusive for the user, so trying to add code with that label from a package results in an error.

2.1.7 Defining relations between hook code

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

voids 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.

There can only be a single relation between two labels for a given hook, i.e., a later `\DeclareHookrule` overwrites any previous declaration.

The ⟨hook⟩ and ⟨label⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

9

\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 2.1.5.

2.1.8 Querying hooks

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

- exist and be empty;

- exist and be non-empty; and

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

Hooks are a bit more complicated: a hook may exist or not, and independently it may or may not be empty. This means that even a hook that doesn't exist may be non-empty and it can also be disabled.

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.

Given that code or rules can be added to a hook even if it doesn't physically exist yet, means that a querying its existence has no real use case (in contrast to other variables that can only be update if they have already been declared). For that reason only the test for emptiness has a public interface.

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. Generic hooks such as `file` and `env` hooks are automatically declared when code is added to them.

\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`) or such code was removed again (via `\RemoveFromHook`), 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.

---

[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).

2.1.9 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`
`\LogHook`

`\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,

- the computed order in which the chunks are executed,

- any code executed on the next invocation only.

`\LogHook` prints the information to the `.log` file, and `\ShowHook` prints them to the terminal/command window and starts TeX's prompt (only in `\errorstopmode`) to wait for user action.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

Suppose a hook `example-hook` whose output of `\ShowHook{example-hook}` is:

```
1    -> The hook 'example-hook':
2    > Code chunks:
3    >     foo -> [code from package 'foo']
4    >     bar -> [from package 'bar']
5    >     baz -> [package 'baz' is here]
6    > Document-level (top-level) code (executed last):
7    >     -> [code from 'top-level']
8    > Extra code for next invocation:
9    >     -> [one-time code]
10   > Rules:
11   >     foo|baz with relation >
12   >     baz|bar with default relation <
13   > Execution order (after applying rules):
14   >     baz, foo, bar.
```

In the listing above, lines 3 to 5 show the three code chunks added to the hook and their respective labels in the format

⟨*label*⟩ `->` ⟨*code*⟩

Line 7 shows the code chunk added by the user in the main document (labeled `top-level`) in the format

`Document-level (top-level) code (executed` ⟨*first|last*⟩`):`
`    ->` ⟨*top-level code*⟩

This code will be either the first or last code executed by the hook (`last` if the hook is normal, `first` if it is reversed). This chunk is not affected by rules and does not take part in sorting.

Line 9 shows the code chunk for the next execution of the hook in the format

```
          -> ⟨next-code⟩
```

This code will be used and disappear at the next `\UseHook{example-hook}`, in contrast to the chunks mentioned earlier, which can only be removed from that hook by doing `\RemoveFromHook{`⟨*label*⟩`}[example-hook]`.

Lines 11 and 12 show the rules declared that affect this hook in the format

⟨*label-1*⟩`|`⟨*label-2*⟩ `with` ⟨*default?*⟩ `relation` ⟨*relation*⟩

which means that the ⟨*relation*⟩ applies to ⟨*label-1*⟩ and ⟨*label-2*⟩, in that order, as detailed in `\DeclareHookRule`. If the relation is `default` it means that this rule applies to ⟨*label-1*⟩ and ⟨*label-2*⟩ in *all* hooks, (unless overridden by a non-default relation).

Finally, line 14 lists the labels in the hook after sorting; that is, in the order they will be executed when the hook is used.

2.1.10 Debugging hook code

`\DebugHooksOn`
`\DebugHooksOff`

`\DebugHooksOn`

Turn the debugging of hook code on or off. This displays most 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_reversed:n`
`\hook_new_pair:nn`

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

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 2.1.5.

`\hook_disable_generic:n`

`\hook_disable_generic:n {`⟨*hook*⟩`}`

Marks `{`⟨*hook*⟩`}` as disabled. Any further attempt to add code to it or declare it, will result in an error and any call to `\hook_use:n` will simply do nothing.

This declaration is intended for use with generic hooks that are known not to work (see `ltcmdhooks-doc`) if they receive code.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\hook_activate_generic:n

\hook_activate_generic:n {⟨*hook*⟩}

This is like \hook_new:n but it does nothing if the hook was previously declared with \hook_new:n. This declaration should be used only in special situations, e.g., when a command from another package needs to be altered and it is not clear whether a generic cmd hook (for that command) has been previously explicitly declared.

Normally \hook_new:n should be used instead of this.

\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.

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 2.1.5.

\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 2.1.5.

\hook_gclear_next_code:n

\hook_gclear_next_code:n {⟨*hook*⟩}

Undo any earlier \hook_gput_next_code:nn.

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

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

If there is no code under the ⟨*label*⟩ in the ⟨*hook*⟩, or if the ⟨*hook*⟩ does not exist, a warning is issued when you attempt to use `\hook_gremove_code:nn`, and the command is ignored.

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 2.1.5.

`\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 2.1.5. 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⟩} {⟨true code⟩} {⟨false code⟩}`
`\hook_if_empty:n`*TF* ⋆	

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_show:n`	`\hook_show:n {⟨hook⟩}`
`\hook_log:n`	

Displays information about the ⟨*hook*⟩ such as

- the code chunks (and their labels) added to it,

- any rules set up to order them,

- the computed order in which the chunks are executed,

- any code executed on the next invocation only.

`\hook_log:n` prints the information to the `.log` file, and `\hook_show:n` prints them to the terminal/command window and starts TEX's prompt (only if `\errorstopmode`) to wait for user action.

The ⟨*hook*⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

`\hook_debug_on:`	`\hook_debug_on:`
`\hook_debug_off:`	

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}
> Document-level (top-level) code (executed last):
>     ---
> Extra code for 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, for 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}
> Document-level (top-level) code (executed last):
>     ---
> Extra code for next invocation:
>     ---
> Rules:
>     packageB|packageA with relation >
> 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}{packageC}
```

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 another `package-too` makes the quotes also in blue and therefore adds:

```
\usepackage{color}
\AddToHook{env/quote/before}[package-too]{\begin{color}{blue}}
\AddToHook{env/quote/after} [package-too]{\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, package-too
```

(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}
>     package-too -> \end {color}
> Document-level (top-level) code (executed first):
```

---

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

```
>       ---
> Extra code for next invocation:
>       ---
> Rules:
>       ---
> Execution order (after reversal):
>     package-too, 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   Difference between "normal" and "one-time" hooks

When executing a hook a developer has the choice of using either \UseHook or \UseOneTimeHook (or their expl3 equivalents \hook_use:n and \hook_use_once:n). This choice affects how \AddToHook is handled after the hook has been executed for the first time.

With normal hooks adding code via \AddToHook means that the code chunk is added to the hook data structure and then used each time \UseHook is called.

With one-time hooks it this is handled slightly differently: After \UseOneTimeHook has been called, any further attempts to add code to the hook via \AddToHook will simply execute the ⟨*code*⟩ immediately.

This has some consequences one needs to be aware of:

- If ⟨*code*⟩ is added to a normal hook after the hook was executed and it is never executed again for one or the other reason, then this new ⟨*code*⟩ will never be executed.

- In contrast if that happens with a one-time hook the ⟨*code*⟩ is executed immediately.

In particular this means that construct such as

    \AddToHook{myhook}
            { ⟨*code-1*⟩ \AddToHook{myhook}{⟨*code-2*⟩} ⟨*code-3*⟩ }

works for one-time hooks[5] (all three code chunks are executed one after another), but it makes little sense with a normal hook, because with a normal hook the first time \UseHook{myhook} is executed it would

- execute ⟨*code-1*⟩,

- then execute \AddToHook{myhook}{code-2} which adds the code chunk ⟨*code-2*⟩ to the hook for use on the next invocation,

- and finally execute ⟨*code-3*⟩.

The second time \UseHook is called it would execute the above and in addition ⟨*code-2*⟩ as that was added as a code chunk to the hook in the meantime. So each time the hook is used another copy of ⟨*code-2*⟩ is added and so that code chunk is executed ⟨*# of invocations*⟩ − 1 times.

---

[5]This is sometimes used with \AtBeginDocument which is why it is supported.

2.6 Generic hooks provided by packages

The hook management system also implements a category of hooks that are called "Generic Hooks". Normally a hook has to be explicitly declared before it can be used in code. This ensures that different packages are not using the same hook name for unrelated purposes—something that would result in absolute chaos. However, there are a number of "standard" hooks where it is unreasonable to declare them beforehand, e.g, each and every command has (in theory) an associated `before` and `after` hook. In such cases, i.e., for command, environment or file hooks, they can be used simply by adding code to them with `\AddToHook`. For more specialized generic hooks, e.g., those provided by babel, you have to additionally enable them with `\ActivateGenericHook` as explained below.

The generic hooks provided by LaTeX are those for `cmd`, `env`, `file`, `include package`, and `class`, and all these are available out of the box: you only have to use `\AddToHook` to add code to them, but you don't have to add `\UseHook` or `\UseOneTimeHook` to your code, because this is already done for you (or, in the case of `cmd` hooks, the command's code is patched at `\begin{document}`, if necessary).

However, if you want to provide further generic hooks in your own code, the situation is slightly different. To do this you should use `\UseHook` or `\UseOneTimeHook`, but *without declaring the hook* with `\NewHook`. As mentioned earlier, a call to `\UseHook` with an undeclared hook name does nothing. So as an additional setup step, you need to explicitly activate your generic hook. Note that a generic hook produced in this way is always a normal hook.

For a truly generic hook, with a variable part in the hook name, such upfront activation would be difficult or impossible, because you typically do not know what kind of variable parts may come up in real documents.

For example, babel may want to provide hooks such as `babel/afterextras/⟨language⟩`. Language support in babel is often done through external language packages. Thus doing the activation for all languages inside the core babel code is not a viable approach. Instead it needs to be done by each language package (or by the user who wants to use a particular hook).

Because the hooks are not declared with `\NewHook` their names should be carefully chosen to ensure that they are (likely to be) unique. Best practice is to include the package or command name, as was done in the babel example above.

Generic hooks defined in this way are always normal hooks (i.e., you can't implement reversed hooks this way). This is a deliberate limitation, because it speeds up the processessing conciderably.

2.7 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@`⟨*hook*⟩ or `\@kernel@after@`⟨*hook*⟩. 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.[6]

2.8 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` (see section 2.1.6) if done outside of a package or class or with the package/class name if called inside such a file (see section 2.1.5).

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 another package's code. When using the optional argument the call is equivalent to running `\AddToHook {begindocument}` [⟨*label*⟩] {⟨*code*⟩}.

`\AtBeginDocument` is a wrapper around the `begindocument` hook (see section 3.2), which is a one-time hook. As such, after the `begindocument` hook is executed at `\begin{document}` any attempt to add ⟨*code*⟩ to this hook with `\AtBeginDocument` or with `\AddToHook` will cause that ⟨*code*⟩ to execute immediately instead. See section 2.5 for more on one-time hooks.

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.

There is also `\AtBeginDvi` which is discussed in conjunction with the shipout hooks.

The few hooks that existed previously in LaTeX 2ε used internally commands such as `\@begindocumenthook` and packages sometimes augmented them directly rather than working through `\AtBeginDocument`. For that reason there is currently support for this, that is, if the system detects that such an internal legacy hook command contains code it adds it to the new hook system under the label `legacy` so that it doesn't get lost.

---

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

However, over time the remaining cases of direct usage need updating because in one of the future release of LaTeX we will turn this legacy support off, as it does unnecessary slow down the processing.

3 LaTeX 2ε commands and environments augmented by hooks

In this section we describe the standard hooks that are now offered by LaTeX, or give pointers to other documents in which they are described. This section will grow over time (and perhaps eventually move to usrguide3).

3.1 Generic hooks

As stated earlier, with the exception of generic hooks, all hooks must be declared with `\NewHook` before they can be used. All generic hooks have names of the form "⟨*type*⟩/⟨*name*⟩/⟨*position*⟩", where ⟨*type*⟩ is from the predefined list shown below, and ⟨*name*⟩ is the variable part whose meaning will depend on the ⟨*type*⟩. The last component, ⟨*position*⟩, has more complex possibilities: it can always be `before` or `after`; for `env` hooks, it can also be `begin` or `end`; and for `include` hooks it can also be `end`. Each specific hook is documented below, or in `ltcmdhooks-doc.pdf` or `ltfilehook-doc.pdf`.

The generic hooks provided by LaTeX belong to one of the six types:

env Hooks executed before and after environments – ⟨*name*⟩ is the name of the environment, and available values for ⟨*position*⟩ are `before`, `begin`, `end`, and `after`;

cmd Hooks added to and executed before and after commands – ⟨*name*⟩ is the name of the command, and available values for ⟨*position*⟩ are `before` and `after`;

file Hooks executed before and after reading a file – ⟨*name*⟩ is the name of the file (with extension), and available values for ⟨*position*⟩ are `before` and `after`;

package Hooks executed before and after loading packages – ⟨*name*⟩ is the name of the package, and available values for ⟨*position*⟩ are `before` and `after`;

class Hooks executed before and after loading classes – ⟨*name*⟩ is the name of the class, and available values for ⟨*position*⟩ are `before` and `after`;

include Hooks executed before and after `\include`d files – ⟨*name*⟩ is the name of the included file (without the `.tex` extension), and available values for ⟨*position*⟩ are `before`, `end`, and `after`.

Each of the hooks above are detailed in the following sections and in linked documentation.

3.1.1 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 hook 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).

env/⟨_env_⟩/after This hook 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.

The 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.

Generic environment hooks are never one-time hooks even with environments that are supposed to appear only once in a document.[7] In contrast to other hooks there is also no need to declare them 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.

Largely for compatibility with existing packages, the following four commands are also available to set the environment hooks; but for new packages we recommend directly using the hook names and \AddToHook.

\BeforeBeginEnvironment

\BeforeBeginEnvironment [⟨_label_⟩] {⟨_env_⟩} {⟨_code_⟩}

This declaration adds to the env/⟨_env_⟩/before hook using the ⟨_label_⟩. If ⟨_label_⟩ is not given, the ⟨_default label_⟩ is used (see section 2.1.5).

\AtBeginEnvironment

\AtBeginEnvironment [⟨_label_⟩] {⟨_env_⟩} {⟨_code_⟩}

This is like \BeforeBeginEnvironment but it adds to the env/⟨_env_⟩/begin hook.

\AtEndEnvironment

\AtEndEnvironment [⟨_label_⟩] {⟨_env_⟩} {⟨_code_⟩}

This is like \BeforeBeginEnvironment but it adds to the env/⟨_env_⟩/end hook.

\AfterEndEnvironment

\AfterEndEnvironment [⟨_label_⟩] {⟨_env_⟩} {⟨_code_⟩}

This is like \BeforeBeginEnvironment but it adds to the env/⟨_env_⟩/after hook.

---

[7]Thus if one adds code to such hooks after the environment has been processed, it will only be executed if the environment appears again and if that doesn't happen the code will never get executed.

3.1.2 Generic hooks for commands

Similar to environments there are now (at least in theory) two generic hooks available for any LaTeX command. These are

cmd/⟨*name*⟩/before This hook is executed at the very start of the command execution.

cmd/⟨*name*⟩/after This hook is executed at the very end of the command body. It is implemented as a reversed hook.

In practice there are restrictions and especially the `after` hook works only with a subset of commands. Details about these restrictions are documented in `ltcmdhooks-doc.pdf` or with code in `ltcmdhooks-code.pdf`.

3.1.3 Generic hooks provided by 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`, `\RequirePackage`, etc. These are documented in `ltfilehook-doc.pdf` or with code in `ltfilehook-code.pdf`.

3.2 Hooks provided by `\begin{document}`

Until 2020 `\begin{document}` offered exactly one hook that one could add to using `\AtBeginDocument`. Experiences over the years have 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 these 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 (all of them one-time hooks):

begindocument/before This hook is executed at the very start of `\document`, one can think of it as a hook for code at the end of the preamble section and this is how it is used by etoolbox's `\AtEndPreamble`.

   This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 2.5).

begindocument This hook is added to when using `\AtBeginDocument` and it 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.

   The hook should not be used to add material for typesetting as we are still in LaTeX's initialization phase and not in the document body. If such material needs to be added to the document body use the next hook instead.

   This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 2.5).

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`.

   This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 2.5).

The generic hooks executed by \begin also exist, i.e., `env/document/before` and `env/document/begin`, but with this special environment it is better use the dedicated one-time hooks above.

3.3   Hooks provided by \end{document}

LaTeX 2_ε 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 (all of them one-time hooks):

enddocument The hook associated with \AtEndDocument. It is immediately called at the beginning of \enddocument.

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.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 2.5).

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, but since there will be no more pages you need to write to it using \immediate\write). 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.).

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 2.5).

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.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 2.5).

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.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 2.5).

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

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 2.5).is it even possible to add code after this one?

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 `\shipout`s is the last one, LaTeX needs to be run several times, so initially it might get executed on the wrong page. See section 3.4 for where to find the details.

It is in also possible to use the generic `env/document/end` hook which is executed by `\end`, i.e., just in front of the first hook above. Note however that the other generic `\end` environment hook, i.e., `env/document/after` will never get executed, because by that time LaTeX has finished the document processing.

3.4 Hooks provided by \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`.

3.5 Hooks provided for paragraphs

The paragraph processing has been augmented to include a number of internal and public hooks. These are documented in `ltpara-doc.pdf` or with code in `ltpara-code.pdf`.

3.6 Hooks provided in NFSS commands

In languages that need to support for more than one script in parallel (and thus several sets of fonts, e.g., supporting both Latin and Japanese fonts), NFSS font commands such as `\sffamily` need to switch both the Latin family to "Sans Serif" and in addition alter a second set of fonts.

To support this, several NFSS commands have hooks to which such support can be added.

rmfamily After `\rmfamily` has done its initial checks and prepared a font series update, this hook is executed before `\selectfont`.

sffamily This is like the `rmfamily` hook, but for the `\sffamily` command.

ttfamily This is like the `rmfamily` hook, but for the `\ttfamily` command.

normalfont The `\normalfont` command resets the font encoding, family, series and shape to their document defaults. It then executes this hook and finally calls `\selectfont`.

expand@font@defaults The internal `\expand@font@defaults` command expands and saves the current defaults for the meta families (rm/sf/tt) and the meta series (bf/md). If the NFSS machinery has been augmented, e.g., for Chinese or Japanese fonts, then further defaults may need to be set at this point. This can be done in this hook which is executed at the end of this macro.

bfseries/defaults, bfseries If the `\bfdefault` was explicitly changed by the user, its new value is used to set the bf series defaults for the meta families (rm/sf/tt) when `\bfseries` is called. The `bfseries/defaults` hook allows further adjustments to be made in this case. This hook is only executed if such a change is detected. In contrast, the `bfseries` hook is always executed just before `\selectfont` is called to change to the new series.

mdseries/defaults, mdseries These two hooks are like the previous ones but they are in the `\mdseries` command.

selectfont This hook is executed inside `\selectfont`, after the current values for *encoding*, *family*, *series*, *shape*, and *size* are evaluated and the new font is selected (and if necessary loaded). After the hook has executed, NFSS will still do any updates necessary for a new *size* (such as changing the size of `\strut`) and any updates necessary to a change in *encoding*.

This hook is intended for use cases where, in parallel to a change in the main font, some other fonts need to be altered (e.g., in CJK processing where you may need to deal with several different alphabets).

4   The Implementation

1 ⟨@@=hook⟩

2 ⟨∗2ekernel | latexrelease⟩
3 `\ExplSyntaxOn`
4 ⟨latexrelease⟩*\NewModuleRelease{2020/10/01}{lthooks}*
5 ⟨latexrelease⟩   *{The~hook~management~system}*

4.1   Debugging

`\g__hook_debug_bool` Holds the current debugging state.

6 `\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`.

7 `\cs_new_eq:NN \__hook_debug:n \use_none:n`
8 `\cs_new_protected:Npn \hook_debug_on:`
9 `  {`
10 `    \bool_gset_true:N \g__hook_debug_bool`
11 `    \__hook_debug_gset:`
12 `  }`
13 `\cs_new_protected:Npn \hook_debug_off:`
14 `  {`
15 `    \bool_gset_false:N \g__hook_debug_bool`
16 `    \__hook_debug_gset:`
17 `  }`

```
18 \cs_new_protected:Npn \__hook_debug_gset:
19   {
20     \cs_gset_protected:Npx \__hook_debug:n ##1
21       { \bool_if:NT \g__hook_debug_bool {##1} }
22   }
```

(*End definition for* \hook_debug_on: *and others. These functions are documented on page* *14.*)

4.2   Borrowing from internals of other kernel modules

__hook_str_compare:nn   Private copy of __str_if_eq:nn

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

(*End definition for* __hook_str_compare:nn.)

4.3   Declarations

\l__hook_tmpa_bool   Scratch boolean used throughout the package.

```
24 \bool_new:N \l__hook_tmpa_bool
```

(*End definition for* \l__hook_tmpa_bool.)

\l__hook_return_tl   Scratch variables used throughout the package.
\l__hook_tmpa_tl
\l__hook_tmpb_tl
```
25 \tl_new:N \l__hook_return_tl
26 \tl_new:N \l__hook_tmpa_tl
27 \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.

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

(*End definition for* \g__hook_all_seq.)

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

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

(*End definition for* \l__hook_cur_hook_tl.)

\l__hook_work_prop   A property list holding a copy of the \g__hook_⟨hook⟩_code_prop of the hook being
sorted to work on, so that changes don't act destructively on the hook data structure.

```
30 \prop_new:N \l__hook_work_prop
```

(*End definition for* \l__hook_work_prop.)

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

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

(*End definition for* \g__hook_used_prop.)

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

```
32 \tl_new:N \g__hook_hook_curr_name_tl
33 \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.

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

(End definition for `__hook_tmp:w.)*

`\tl_gremove_once:Nx` Some variants of expl3 functions.
`\tl_show:x`
`\tl_log:x`
> *FMi: should probably be moved to expl3*

```
35 \cs_generate_variant:Nn \tl_gremove_once:Nn { Nx }
36 \cs_generate_variant:Nn \tl_show:n { x }
37 \cs_generate_variant:Nn \tl_log:n { x }
```

(End definition for `\tl_gremove_once:Nx`, `\tl_show:x`, *and* `\tl_log:x.)*

`\s__hook_mark` Scan mark used for delimited arguments.

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

(End definition for `\s__hook_mark.)*

`\__hook_clean_to_scan:w` Removes tokens until the next `\s__hook_mark`.

```
39 \cs_new:Npn \__hook_clean_to_scan:w #1 \s__hook_mark { }
```

(End definition for `__hook_clean_to_scan:w.)*

`\__hook_tl_set:Nn` Private copies of a few expl3 functions. l3debug will only add debugging to the public
`\__hook_tl_set:Nx` names, not to these copies, so we don't have to use `\debug_suspend:` and `\debug_-`
`\__hook_tl_set:cn` `resume:` everywhere.
`\__hook_tl_set:cx`   Functions like `\__hook_tl_set:Nn` have to be redefined, rather than copied because
in expl3 they use `\__kernel_tl_(g)set:Nx`, which is also patched by l3debug.

```
40 \cs_new_protected:Npn \__hook_tl_set:Nn #1#2
41   { \cs_set_nopar:Npx #1 { \__kernel_exp_not:w {#2} } }
42 \cs_new_protected:Npn \__hook_tl_set:Nx #1#2
43   { \cs_set_nopar:Npx #1 {#2} }
44 \cs_generate_variant:Nn \__hook_tl_set:Nn { c }
45 \cs_generate_variant:Nn \__hook_tl_set:Nx { c }
```

(End definition for `__hook_tl_set:Nn.)*

`\__hook_tl_gset:Nn` Same as above.
`\__hook_tl_gset:No`
`\__hook_tl_gset:Nx`
`\__hook_tl_gset:cn`
`\__hook_tl_gset:co`
`\__hook_tl_gset:cx`

```
46 \cs_new_protected:Npn \__hook_tl_gset:Nn #1#2
47   { \cs_gset_nopar:Npx #1 { \__kernel_exp_not:w {#2} } }
48 \cs_new_protected:Npn \__hook_tl_gset:No #1#2
49   { \cs_gset_nopar:Npx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
50 \cs_new_protected:Npn \__hook_tl_gset:Nx #1#2
51   { \cs_gset_nopar:Npx #1 {#2} }
52 \cs_generate_variant:Nn \__hook_tl_gset:Nn { c }
53 \cs_generate_variant:Nn \__hook_tl_gset:No { c }
54 \cs_generate_variant:Nn \__hook_tl_gset:Nx { c }
```

(End definition for `__hook_tl_gset:Nn.)*

`\__hook_tl_gput_right:Nn`
`\__hook_tl_gput_right:No`
`\__hook_tl_gput_right:cn`

Same as above.

```
55 \cs_new_protected:Npn \__hook_tl_gput_right:Nn #1#2
56   { \__hook_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
57 \cs_generate_variant:Nn \__hook_tl_gput_right:Nn { No, cn }
```

(*End definition for* `\__hook_tl_gput_right:Nn`.)

`\__hook_tl_gput_left:Nn`
`\__hook_tl_gput_left:No`

Same as above.

```
58 \cs_new_protected:Npn \__hook_tl_gput_left:Nn #1#2
59   {
60     \__hook_tl_gset:Nx #1
61       { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
62   }
63 \cs_generate_variant:Nn \__hook_tl_gput_left:Nn { No }
```

(*End definition for* `\__hook_tl_gput_left:Nn`.)

`\__hook_tl_gset_eq:NN`

Same as above.

```
64 \cs_new_eq:NN \__hook_tl_gset_eq:NN \tl_gset_eq:NN
```

(*End definition for* `\__hook_tl_gset_eq:NN`.)

`\__hook_tl_gclear:N`
`\__hook_tl_gclear:c`

Same as above.

```
65 \cs_new_protected:Npn \__hook_tl_gclear:N #1
66   { \__hook_tl_gset_eq:NN #1 \c_empty_tl }
67 \cs_generate_variant:Nn \__hook_tl_gclear:N { c }
```

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

4.4 Providing new hooks

4.4.1 The data structures of a hook

`\g_@@_⟨hook⟩_code_prop`
`\@@_␣⟨hook⟩`
`\@@_next_␣⟨hook⟩`

Hooks have a name (called ⟨*hook*⟩ in the description below) and for each hook we have to provide a number of data structures. These are

`\g__hook_⟨hook⟩_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_⟨hook⟩_rule_⟨label1⟩|⟨label2⟩_tl` A token list holding the relation between ⟨*label1*⟩ and ⟨*label2*⟩ in the ⟨*hook*⟩. The ⟨*labels*⟩ are lexically (reverse) sorted to ensure that two labels always point to the same token list. For global rules, the ⟨*hook*⟩ name is ??.

`\__hook_␣⟨hook⟩` 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. This token list is named like that so that in case of an error inside the hook, the reported token list in the error is shorter, and to make it simpler to normalize hook names in `\__hook_make_name:n`.

`\g__hook_⟨hook⟩_reversed_tl` Some hooks are "reversed". This token list stores a - for such hook so that it can be identified. The - character is used because ⟨*reversed*⟩1 is +1 for normal hooks and −1 for reversed ones.

28

`\g__hook_`⟨*hook*⟩`_declared_tl` This token list serves as marker for the hook being officially declared. Its existence is tested to raise an error in case another declaration is attempted.

`\__hook_toplevel␣`⟨*hook*⟩ This token list stores the code inserted in the hook from the user's document, in the `top-level` label. This label is special, and doesn't participate in sorting. Instead, all code is appended to it and executed after (or before, if the hook is reversed) the normal hook code, but before the `next` code chunk.

`\__hook_next␣`⟨*hook*⟩ 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. This token list follows the same naming scheme than the main `\__hook␣`⟨*hook*⟩ token list. It is called `\__hook_next␣`⟨*hook*⟩ rather than `\__hook␣next_`⟨*hook*⟩ because otherwise a hook whose name is `next_`⟨*hook*⟩ would clash with the next code-token list of the hook called ⟨*hook*⟩.

4.4.2 On the existence of hooks

A hook may be in different states of existence. Here we give an overview of the internal commands to set up hooks and explain how the different states are distinguished. The actual implementation then follows in subsequent sections.

One problem we have to solve is that we need to be able to add code to hooks (e.g., with `\AddToHook`) even if that code has not yet been declared. For example, one package needs to write into a hook of another package, but that package may not get loaded, or is loaded only later. Another problem is that most hooks, but not the generic hooks, require a declaration.

We therefore distinguish the following states for a hook, which are managed by four different tests: structure existence (`\__hook_if_structure_exist:nTF`), creation (`\__hook_if_usable:nTF`), declaration (`\__hook_if_declared:nTF`) and disabled or not (`\__hook_if_disabled:nTF`)

not existing Nothing is known about the hook so far. This state can be detected with `\__hook_if_structure_exist:nTF` (which uses the false branch).

In this state the hook can be declared, disabled, rules can be defined or code could be added to it, but it is not possible to use the hook (with `\UseHook`).

basic data structure set up A hook is this state when its basic data structure has been set up (using `\__hook_init_structure:n`). The data structure setup happens automatically when commands such as `\AddToHook` are used and the hook is at that point in state "not existing".

In this state the four tests give the following results:

`\__hook_if_structure_exist:nTF` returns `true`.

`\__hook_if_usable:nTF` returns `false`.

`\__hook_if_declared:nTF` returns `false`.

`\__hook_if_disabled:nTF` returns `false`.

The allowed actions are the same as in the "not existing" state.

declared A hook is in this state it is not disabled and was explicitly declared (e.g., with `\NewHook`). In this case the four tests give the following results:

`\__hook_if_structure_exist:nTF` returns `true`.

`\__hook_if_usable:nTF` returns `true`.

`\__hook_if_declared:nTF` returns `true`.

`\__hook_if_disabled:nTF` returns `false`.

usable A hook is in this state if it is not disabled, was not explicitly declared but nevertheless is allowed to be used (with `\UseHook` or `\hook_use:n`). This state is only possible for generic hooks as they do not need to be declared. Therefore such hooks move directly from state "not existing" to "usable" the moment a declaration such as `\AddToHook` wants to add to the hook data structure. In this state the tests give the following results:

`\__hook_if_structure_exist:nTF` returns `true`.

`\__hook_if_usable:nTF` returns `true`.

`\__hook_if_declared:nTF` returns `false`.

`\__hook_if_disabled:nTF` returns `false`.

disabled A generic hook in any state is moved to this state when `\DisableGenericHook` is used. This changes the tests to give the following results:

`\__hook_if_structure_exist:nTF` *unchanged*.

`\__hook_if_usable:nTF` returns `false`.

`\__hook_if_declared:nTF` returns `true`.

`\__hook_if_disabled:nTF` returns `true`.

The structure test is unchanged (if the hook was unknown before it is `false`, otherwise `true`). The usable test returns `false` so that any `\UseHook` will bypass the hook from now on. The declared test returns true so that any further `\NewHook` generates an error and the disabled test returns true so that `\AddToHook` can return an error.

FMi: maybe it should do this only after begin document?

4.4.3 Setting hooks up

`\hook_new:n`
`\__hook_new:n`

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

```
68 \cs_new_protected:Npn \hook_new:n #1
69   { \__hook_normalize_hook_args:Nn \__hook_new:n {#1} }
70 \cs_new_protected:Npn \__hook_new:n #1
71   {
```

We check if the hook was already *explicitly* declared with `\hook_new:n`, and if it already exists we complain, otherwise set the "created" flag for the hook so that it errors next time `\hook_new:n` is used.

```
72     \__hook_if_declared:nTF {#1}
73       { \msg_error:nnn { hooks } { exists } {#1} }
```

```
74        {
75          \tl_new:c { g__hook_#1_declared_tl }
76          \__hook_make_usable:n {#1}
77        }
78    }
```

(*End definition for* \hook_new:n *and* __hook_new:n. *This function is documented on page* *12.*)

__hook_make_usable:n   This initializes all hook data structures for the hook but if used on its own doesn't mark the hook as declared (as \hook_new:n does, so a later \hook_new:n on that hook will not result in an error. This command is internally used by \hook_gput_code:n when adding code to a generic hook.

```
79  \cs_new_protected:Npn \__hook_make_usable:n #1
80    {
```

Now we check if the hook's data structure can be safely created without expl3 raising errors, then we add the hook name to the list of all hooks and allocate the necessary data structures for the new hook, otherwise just do nothing.

```
81    \tl_if_exist:cF { __hook~#1 }
82      {
83        \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:

```
84        \tl_new:c { __hook~#1 }
```

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

```
85        \__hook_init_structure:n {#1}
```

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

```
86        \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.

```
87        \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$_\varepsilon$ hooks of the form \@...hook, e.g., \@begindocumenthook. there have been a few of them and they have been added to using \g@addto@macro. If there exists such a macro matching the name of the new hook, i.e., \@⟨*hook-name*⟩hook and it is not empty then we add its contents as a code chunk under the label legacy.

Warning: this support will vanish in future releases!

```
88        \__hook_include_legacy_code_chunk:n {#1}
89      }
90    }
```

(*End definition for* __hook_make_usable:n.)

__hook_init_structure:n   This function declares the basic data structures for a hook without explicit declaring the hook itself. This is needed to allow adding to undeclared hooks. Here it is unnecessary to check whether all variables exist, since all three are declared at the same time (either all of them exist, or none).

It creates the hook code pool (\g__hook_⟨*hook*⟩_code_prop) and the top-level and next token lists. A hook is initialized with __hook_init_structure:n the first

time anything is added to it. Initializing a hook just with `\__hook_init_structure:n` will not make it usable with `\hook_use:n`.

```
91 \cs_new_protected:Npn \__hook_init_structure:n #1
92   {
93     \__hook_if_structure_exist:nF {#1}
94       {
95         \prop_new:c { g__hook_#1_code_prop }
96         \tl_new:c { __hook_toplevel~#1 }
97         \tl_new:c { __hook_next~#1 }
98       }
99   }
```

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

`\hook_new_reversed: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.

```
100 \cs_new_protected:Npn \hook_new_reversed:n #1
101   { \__hook_normalize_hook_args:Nn \__hook_new_reversed:n {#1} }
102 \cs_new_protected:Npn \__hook_new_reversed:n #1
103   {
104     \__hook_new:n {#1}
```

If the hook already exists the above will generate an error message, so the next line should be executed (but it is — too bad).

```
105     \tl_gset:cn { g__hook_#1_reversed_tl } { - }
106   }
```

(*End definition for* `\hook_new_reversed:n` *and* `\__hook_new_reversed:n`. *This function is documented on page 12.*)

`\hook_new_pair:nn`

A shorthand for declaring a normal and a (matching) reversed hook in one go.

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

(*End definition for* `\hook_new_pair:nn`. *This function is documented on page 12.*)

`\__hook_include_legacy_code_chunk:n`

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

If this macro is not empty we add it under the label `legacy` to the current hook and then empty it globally. This way packages or classes directly manipulating commands such as `\@begindocumenthook` still get their hook data added.

Warning: this support will vanish in future releases!

```
109 \cs_new_protected:Npn \__hook_include_legacy_code_chunk:n #1
110   {
```

If the macro doesn't exist (which is the usual case) then nothing needs to be done.

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

Of course if the legacy hook exists but is empty, there is no need to add anything under `legacy` the legacy label.

```
112       {
113         \tl_if_empty:cF { @#1hook }
114           {
115             \exp_args:Nnnv \__hook_hook_gput_code_do:nnn {#1}
116                                 { legacy } { @#1hook }
```

Once added to the hook, we need to clear it otherwise it might get added again later if the hook data gets updated.

```
117                    \__hook_tl_gclear:c { @#1hook }
118                }
119            }
120        }
```

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

4.4.4 Disabling and providing hooks

`\hook_disable_generic:n`
`\__hook_disable:n`
`\__hook_if_disabled_p:n`
`\__hook_if_disabled:nTF`

Disables a hook by creating its `\g__hook_⟨hook⟩_declared_tl` so that the hook errors when used with `\hook_new:n`, then it undefines `\__hook␣⟨hook⟩` so that it may not be executed.

This does not clear any code that may be already stored in the hook's structure, but doesn't allow adding more code. `\__hook_if_disabled:nTF` uses that specific combination to check if the hook is disabled.

```
121 ⟨latexrelease⟩\IncludeInRelease{2021/06/01}%
122 ⟨latexrelease⟩                {\hook_disable_generic:n}{Disable~hooks}
123 \cs_new_protected:Npn \hook_disable_generic:n #1
124    { \__hook_normalize_hook_args:Nn \__hook_disable:n {#1} }
125 \cs_new_protected:Npn \__hook_disable:n #1
126    {
127       \tl_gclear_new:c { g__hook_#1_declared_tl }
128       \cs_undefine:c { __hook~#1 }
129    }
130 \prg_new_conditional:Npnn \__hook_if_disabled:n #1 { p, T, F, TF }
131    {
132       \bool_lazy_and:nnTF
133          { \tl_if_exist_p:c { g__hook_#1_declared_tl } }
134          { ! \tl_if_exist_p:c { __hook~#1 } }
135       { \prg_return_true: }
136       { \prg_return_false: }
137    }
138 ⟨latexrelease⟩\EndIncludeInRelease

139 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}
140 ⟨latexrelease⟩                         {\hook_disable_generic:n}{Disable~hooks}
141 ⟨latexrelease⟩
142 ⟨latexrelease⟩\cs_new_protected:Npn \hook_disable_generic:n #1 {}
143 ⟨latexrelease⟩
144 ⟨latexrelease⟩\EndIncludeInRelease
```

(*End definition for* `\hook_disable_generic:n, \__hook_disable:n, and \__hook_if_disabled:nTF.` *This function is documented on page 12.*)

`\hook_activate_generic:n`
`\__hook_activate_generic:n`

The `\hook_activate_generic:n` declaration declares a new hook if it wasn't declared already, in which case it only checks that the already existing hook is not a reversed hook.

```
145 ⟨latexrelease⟩\IncludeInRelease{2021/06/01}%
146 ⟨latexrelease⟩                {\hook_activate_generic:n}{Providing~hooks}
147 \cs_new_protected:Npn \hook_activate_generic:n #1
148    { \__hook_normalize_hook_args:Nn \__hook_activate_generic:nn {#1} {   } }
```

```
149 \cs_new_protected:Npn \__hook_activate_generic:nn #1 #2
150   {
```

If the hook to be activated was disabled we warn (for now — this may change).

```
151     \__hook_if_disabled:nTF {#1}
152       { \msg_warning:nnn { hooks } { activate-disabled } {#1} }
```

Otherwise we check if the hook is not declared, and if it isn't, figure out if it's reversed or not, then declare it accordingly.

```
153       {
154         \__hook_if_declared:nF {#1}
155           {
156             \tl_new:c { g__hook_#1_declared_tl }
157             \__hook_make_usable:n {#1}
158             \tl_gset:cx { g__hook_#1_reversed_tl }
159               { \__hook_if_generic_reversed:nT {#1} { - } }
160           }
161       }
162   }
```

(*End definition for* \hook_activate_generic:n *and* __hook_activate_generic:n. *This function is documented on page* *13.*)

```
163 ⟨latexrelease⟩\EndIncludeInRelease
164 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}
165 ⟨latexrelease⟩                        {\hook_activate_generic:n}{Providing~hooks}
166 ⟨latexrelease⟩
167 ⟨latexrelease⟩\cs_new_protected:Npn \hook_activate_generic:n #1 {}
168 ⟨latexrelease⟩
169 ⟨latexrelease⟩\EndIncludeInRelease
```

4.5   Parsing a label

__hook_parse_label_default:n 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 by __hook_currname_or_default:.

```
170 \cs_new:Npn \__hook_parse_label_default:n #1
171   {
172     \tl_if_novalue:nTF {#1}
173       { \__hook_currname_or_default: }
174       { \tl_trim_spaces_apply:nN {#1} \__hook_parse_dot_label:n }
175   }
```

(*End definition for* __hook_parse_label_default:n.)

__hook_parse_dot_label:n
__hook_parse_dot_label:w
__hook_parse_dot_label_cleanup:w
__hook_parse_dot_label_aux:w

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:. Otherwise the label is returned unchanged.

```
176 \cs_new:Npn \__hook_parse_dot_label:n #1
177   {
178     \tl_if_empty:nTF {#1}
179       {
180         \msg_expandable_error:nn { hooks } { empty-label }
181         \__hook_currname_or_default:
182       }
```

34

```
183        {
184          \str_if_eq:nnTF {#1} { . }
185            { \__hook_currname_or_default: }
186            { \__hook_parse_dot_label:w #1 ./ \s__hook_mark }
187        }
188    }
189 \cs_new:Npn \__hook_parse_dot_label:w #1 ./ #2 \s__hook_mark
190    {
191      \tl_if_empty:nTF {#1}
192        { \__hook_parse_dot_label_aux:w #2 \s__hook_mark }
193        {
194          \tl_if_empty:nTF {#2}
195            { \__hook_make_name:n {#1} }
196            { \__hook_parse_dot_label_cleanup:w #1 ./ #2 \s__hook_mark }
197        }
198    }
199 \cs_new:Npn \__hook_parse_dot_label_cleanup:w #1 ./ \s__hook_mark {#1}
200 \cs_new:Npn \__hook_parse_dot_label_aux:w #1 ./ \s__hook_mark
201    { \__hook_currname_or_default: / \__hook_make_name:n {#1} }
```

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

`\__hook_currname_or_default:`   This uses `\g__hook_hook_curr_name_tl` if it is set, otherwise it tries `\@currname`. If neither is set, it raises an error and uses the fallback value `label-missing`.

```
202 \cs_new:Npn \__hook_currname_or_default:
203    {
204      \tl_if_empty:NTF \g__hook_hook_curr_name_tl
205        {
206          \tl_if_empty:NTF \@currname
207            {
208              \msg_expandable_error:nnn { latex2e } { should-not-happen }
209                { Empty~default~label. }
210              \__hook_make_name:n { label-missing }
211            }
212            { \@currname }
213        }
214        { \g__hook_hook_curr_name_tl }
215    }
```

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

`\__hook_make_name:n`
`\__hook_make_name:w`   This provides a standard sanitization of a hook's name. It uses `\cs:w` to build a control sequence out of the hook name, then uses `\cs_to_str:N` to get the string representation of that, without the escape character. `\cs:w`-based expansion is used instead of e-based because Unicode characters don't behave well inside `\expanded`. The macro adds the `\_-hook␣` prefix to the hook name to reuse the hook's code token list to build the csname and avoid leaving "public" control sequences defined (as `\relax`) in TeX's memory.

```
216 \cs_new:Npn \__hook_make_name:n #1
217    {
218      \exp_after:wN \exp_after:wN \exp_after:wN \__hook_make_name:w
219      \exp_after:wN \token_to_str:N \cs:w __hook~ #1 \cs_end:
220    }
221 \exp_last_unbraced:NNNNo
222 \cs_new:Npn \__hook_make_name:w #1 \tl_to_str:n { __hook~ } { }
```

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

<div style="float:left">

`\__hook_normalize_hook_args:Nn`
`\__hook_normalize_hook_args:Nnn`
`\__hook_normalize_hook_rule_args:Nnnnn`
`\__hook_normalize_hook_args_aux:Nn`

</div>

This is the standard route for normalizing hook and label arguments. The main macro does the entire operation within a group so that csnames made by `\__hook_make_-name:n` are wiped off before continuing. This means that this function cannot be used for `\hook_use:n`!

```
223 \cs_new_protected:Npn \__hook_normalize_hook_args_aux:Nn #1 #2
224   {
225     \group_begin:
226     \use:e
227       {
228         \group_end:
229         \exp_not:N #1 #2
230       }
231   }
232 \cs_new_protected:Npn \__hook_normalize_hook_args:Nn #1 #2
233   {
234     \__hook_normalize_hook_args_aux:Nn #1
235       { { \__hook_parse_label_default:n {#2} } }
236   }
237 \cs_new_protected:Npn \__hook_normalize_hook_args:Nnn #1 #2 #3
238   {
239     \__hook_normalize_hook_args_aux:Nn #1
240       {
241         { \__hook_parse_label_default:n {#2} }
242         { \__hook_parse_label_default:n {#3} }
243       }
244   }
245 \cs_new_protected:Npn \__hook_normalize_hook_rule_args:Nnnnn #1 #2 #3 #4 #5
246   {
247     \__hook_normalize_hook_args_aux:Nn #1
248       {
249         { \__hook_parse_label_default:n {#2} }
250         { \__hook_parse_label_default:n {#3} }
251         { \tl_trim_spaces:n {#4} }
252         { \__hook_parse_label_default:n {#5} }
253       }
254   }
```

(*End definition for* `\__hook_normalize_hook_args:Nn` *and others.*)

<div style="float:left">

`\__hook_curr_name_push:n`
`\__hook_curr_name_push_aux:n`
`\__hook_curr_name_pop:`
`\__hook_end_document_label_check:`

</div>

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

To pull that one 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 (this stack should never go empty). If we're building the format, set the default label to be top-level:

```
255 \tl_gset:Nn \g__hook_hook_curr_name_tl { top-level }
```

Then, in case we're in latexrelease we push something on the stack to support roll forward. But in some rare cases, latexrelease may be loaded inside another package (notably platexrelease), so we'll first push the top-level entry:

256 ⟨latexrelease⟩\seq_if_empty:NT \g__hook_name_stack_seq
257 ⟨latexrelease⟩ { \seq_gput_right:Nn \g__hook_name_stack_seq { top-level } }

then we dissect the \@currnamestack, adding \@currname to the stack:

258 ⟨latexrelease⟩\cs_set_protected:Npn __hook_tmp:w #1 #2 #3
259 ⟨latexrelease⟩ {
260 ⟨latexrelease⟩   \quark_if_recursion_tail_stop:n {#1}
261 ⟨latexrelease⟩   \seq_gput_right:Nn \g__hook_name_stack_seq {#1}
262 ⟨latexrelease⟩   __hook_tmp:w
263 ⟨latexrelease⟩ }
264 ⟨latexrelease⟩\exp_after:wN __hook_tmp:w \@currnamestack
265 ⟨latexrelease⟩ \q_recursion_tail \q_recursion_tail
266 ⟨latexrelease⟩ \q_recursion_tail \q_recursion_stop

and finally set the default label to be the \@currname:

267 ⟨latexrelease⟩\tl_gset:Nx \g__hook_hook_curr_name_tl { \@currname }
268 ⟨latexrelease⟩\seq_gpop_right:NN \g__hook_name_stack_seq \l__hook_tmpa_tl

Two commands keep track of the stack: when a file is input, __hook_curr_name_-push:n pushes the current default label onto the stack and sets the new default label (all in one go):

269 \cs_new_protected:Npn __hook_curr_name_push:n #1
270   { \exp_args:Nx __hook_curr_name_push_aux:n { __hook_make_name:n {#1} } }
271 \cs_new_protected:Npn __hook_curr_name_push_aux:n #1
272   {
273   \tl_if_blank:nTF {#1}
274   { \msg_error:nn { hooks } { no-default-label } }
275   {
276   \str_if_eq:nnTF {#1} { top-level }
277   {
278   \msg_error:nnnnn { hooks } { set-top-level }
279   { to } { PushDefaultHookLabel } {#1}
280   }
281   {
282   \seq_gpush:NV \g__hook_name_stack_seq \g__hook_hook_curr_name_tl
283   \tl_gset:Nn \g__hook_hook_curr_name_tl {#1}
284   }
285   }
286   }

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

287 \cs_new_protected:Npn __hook_curr_name_pop:
288   {
289   \seq_gpop:NNTF \g__hook_name_stack_seq \l__hook_return_tl
290   { \tl_gset_eq:NN \g__hook_hook_curr_name_tl \l__hook_return_tl }
291   { \msg_error:nn { hooks } { extra-pop-label } }
292   }

At the end of the document we want to check if there was no __hook_curr_name_-push:n without a matching __hook_curr_name_pop: (not a critical error, but it might indicate that something else is not quite right):

293 \tl_gput_right:Nn \@kernel@after@enddocument@afterlastpage
294   { __hook_end_document_label_check: }

37

```
295 \cs_new_protected:Npn \__hook_end_document_label_check:
296   {
297     \seq_gpop:NNT \g__hook_name_stack_seq \l__hook_return_tl
298       {
299         \msg_error:nnx { hooks } { missing-pop-label }
300           { \g__hook_hook_curr_name_tl }
301         \tl_gset_eq:NN \g__hook_hook_curr_name_tl \l__hook_return_tl
302         \__hook_end_document_label_check:
303       }
304   }
```

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.

```
305 \cs_new_protected:Npn \__hook_set_default_hook_label:n #1
306   {
307     \seq_if_empty:NTF \g__hook_name_stack_seq
308       {
309         \msg_error:nnnnn { hooks } { set-top-level }
310           { for } { SetDefaultHookLabel } {#1}
311       }
312       { \exp_args:Nx \__hook_set_default_label:n { \__hook_make_name:n {#1} } }
313   }
314 \cs_new_protected:Npn \__hook_set_default_label:n #1
315   {
316     \str_if_eq:nnTF {#1} { top-level }
317       {
318         \msg_error:nnnnn { hooks } { set-top-level }
319           { to } { SetDefaultHookLabel } {#1}
320       }
321       { \tl_gset:Nn \g__hook_hook_curr_name_tl {#1} }
322   }
```

(*End definition for* `\__hook_curr_name_push:n` *and others.*)

4.6   Adding or removing hook code

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

```
323 \cs_new_protected:Npn \hook_gput_code:nnn #1 #2
324   { \__hook_normalize_hook_args:Nnn \__hook_gput_code:nnn {#1} {#2} }
325 \cs_new_protected:Npn \__hook_gput_code:nnn #1 #2 #3
326   {
```

First check if the code should be executed immediately, rather than stored:

```
327     \__hook_if_execute_immediately:nTF {#1}
328       {#3}
329       {
```

Then check if the hook is usable.

```
330         \__hook_if_usable: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.

```
331            {
332                \__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.

```
333                \__hook_update_hook_code:n {#1}
334            }
```

If the hook is not usable, before giving up, check if it's not disabled and otherwise try to declare it as a generic hook, if its name matches one of the valid patterns.

```
335            {
336                \__hook_if_disabled:nTF {#1}
337                  { \msg_error:nnn { hooks } { hook-disabled } {#1} }
338                  { \__hook_try_declaring_generic_hook:nnn {#1} {#2} {#3} }
339            }
340        }
341    }
```

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

```
342 \cs_new_protected:Npn \__hook_hook_gput_code_do:nnn #1 #2 #3
343    {
```

However, first some debugging info if debugging is enabled:

```
344      \__hook_debug:n{\iow_term:x{****~ Add~ to~
345                      \__hook_if_usable:nF {#1} { undeclared~ }
346                      hook~ #1~ (#2)
347                      \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. If the current label is top-level, the code is added to a dedicated token list __hook_toplevel␣⟨hook⟩ that goes at the end of the hook (or at the beginning, for a reversed hook), just before __hook_next␣⟨hook⟩.

```
348      \str_if_eq:nnTF {#2} { top-level }
349        {
350          \str_if_eq:eeTF { top-level } { \__hook_currname_or_default: }
351            {
```

If the hook's basic structure does not exist, we need to declare it with __hook_init_-structure:n.

```
352              \__hook_init_structure:n {#1}
353              \__hook_tl_gput_right:cn { __hook_toplevel~#1 } {#3}
354            }
355            { \msg_error:nnn { hooks } { misused-top-level } {#1} }
356        }
357        {
358          \prop_get:cnNTF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
359            {
360              \prop_gput:cno { g__hook_#1_code_prop } {#2}
361                { \l__hook_return_tl #3 }
362            }
363            { \prop_gput:cnn { g__hook_#1_code_prop } {#2} {#3} }
364        }
365    }
```

(*End definition for* `\hook_gput_code:nnn`*,* `\__hook_gput_code:nnn`*, and* `\__hook_hook_gput_code_-`
`do:nnn`*. This function is documented on page 13.*)

`\__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.

```
366 \cs_new_protected:Npn \__hook_gput_undeclared_hook:nnn #1 #2 #3
367   {
368     \__hook_init_structure:n {#1}
369     \__hook_hook_gput_code_do:nnn {#1} {#2} {#3}
370   }
```

(*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.

```
371 ⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_try_declaring_generic_hook:nnn}
372 ⟨latexrelease⟩                        {Standardise~generic~hook~names}
373 \cs_new_protected:Npn \__hook_try_declaring_generic_hook:nnn #1
374   {
375     \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
376       \hook_gput_code:nnn
377       \__hook_gput_undeclared_hook:nnn
378         {#1}
379   }
380 \cs_new_protected:Npn \__hook_try_declaring_generic_next_hook:nn #1
381   {
382     \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
383       \hook_gput_next_code:nn
384       \__hook_gput_next_do:nn
385         {#1}
386   }
387 ⟨latexrelease⟩\EndIncludeInRelease
388 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_try_declaring_generic_hook:nnn}
389 ⟨latexrelease⟩                        {Standardise~generic~hook~names}
390 ⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_declaring_generic_hook:nnn #1
391 ⟨latexrelease⟩  {
392 ⟨latexrelease⟩    \__hook_try_declaring_generic_hook:nNNnn {#1}
393 ⟨latexrelease⟩      \hook_gput_code:nnn \__hook_gput_undeclared_hook:nnn
394 ⟨latexrelease⟩  }
395 ⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_declaring_generic_next_hook:nn #1
396 ⟨latexrelease⟩  {
397 ⟨latexrelease⟩    \__hook_try_declaring_generic_hook:nNNnn {#1}
398 ⟨latexrelease⟩      \hook_gput_next_code:nn \__hook_gput_next_do:nn
399 ⟨latexrelease⟩  }
```

(End definition for `\__hook_try_declaring_generic_hook:nnn` *and* `\__hook_try_declaring_generic_-` *)*
next_hook:nn.)

`\__hook_try_declaring_generic_hook:nNNnn`
`__hook_try_declaring_generic_hook_split:nNNnn`

`\__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.

```
400 ⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_declaring_generic_hook:nNNnn #1
401 ⟨latexrelease⟩  {
402 ⟨latexrelease⟩    \__hook_if_file_hook:wTF #1 / \s__hook_mark
403 ⟨latexrelease⟩      {
404 ⟨latexrelease⟩        \exp_args:Ne \__hook_try_declaring_generic_hook_split:nNNnn
405 ⟨latexrelease⟩          { \exp_args:Ne \__hook_file_hook_normalize:n {#1} }
406 ⟨latexrelease⟩      }
407 ⟨latexrelease⟩      { \__hook_try_declaring_generic_hook_split:nNNnn {#1} }
408 ⟨latexrelease⟩  }
409 ⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_declaring_generic_hook_split:nNNnn #1 #2 #3
410 ⟨latexrelease⟩  {
411 ⟨latexrelease⟩    \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
412 ⟨latexrelease⟩      { #2 }
413 ⟨latexrelease⟩      { #3 } {#1}
414 ⟨latexrelease⟩  }
415 ⟨latexrelease⟩\EndIncludeInRelease
```

(End definition for `\__hook_try_declaring_generic_hook:nNNnn` *and* `\__hook_try_declaring_generic_-` *)*
hook_split:nNNnn.)

`\__hook_try_declaring_generic_hook:wnTF`

```
416 ⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_try_declaring_generic_hook:wn}%
417 ⟨latexrelease⟩                    {Standardise~generic~hook~names}
418 \prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
419   #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
420   {
421     \__hook_if_generic:nTF {#5}
422       {
423         \__hook_if_usable:nF {#5}
424           {
```

If the hook doesn't exist yet we check if it is a `cmd` hook and if so we attempt patching the command in addition to declaring the hook.

For some commands this will not be possible, in which case `\__hook_patch_cmd_-or_delay:Nnn` (defined in `ltcmdhooks`) will generate an appropriate error message.

```
425             \str_if_eq:nnT {#1} { cmd }
426               { \__hook_try_put_cmd_hook:n {#5} }
```

Declare the hook always even if it can't really be used (error message generated elsewhere).

Here we use `\__hook_make_usable:n`, so that a `\hook_new:n` is still possible later.

```
427             \__hook_make_usable:n {#5}
428           }
429         \__hook_if_generic_reversed:nT {#5}
430           { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
431         \prg_return_true:
```

41

```
432          }
433          {
```

Generic hooks are all named ⟨*type*⟩/⟨*name*⟩/⟨*place*⟩, where ⟨*type*⟩ and ⟨*place*⟩ are predefined (`\c__hook_generic_`⟨*type*⟩`/./`⟨*place*⟩`_tl`), and ⟨*name*⟩ is the variable component. Older releases had some hooks with the ⟨*name*⟩ in the third part, so the code below supports that syntax for a while, with a warning.

The `\exp_after:wN ... \exp:w` trick is there to remove the conditional structure inserted by `\__hook_try_declaring_generic_hook:wnTF` and thus allow access to the tokens that follow it, as is needed to keep things going.

When the deprecation cycle ends, the lines below should all be replaced by `\prg_-return_false:`.

```
434          \__hook_if_deprecated_generic:nTF {#5}
435            {
436              \__hook_deprecated_generic_warn:n {#5}
437              \exp_after:wN \__hook_declare_deprecated_generic:NNn
438              \exp:w % \exp_end:
439            }
440            { \prg_return_false: }
441        }
442    }
```

`\__hook_deprecated_generic_warn:Nn`
`\__hook_deprecated_generic_warn:Nw`

`\__hook_deprecated_generic_warn:n` will issue a deprecation warning for a given hook, and mark that hook such that the warning will not be issued again (multiple warnings can be issued, but only once per hook).

```
443 \cs_new_protected:Npn \__hook_deprecated_generic_warn:n #1
444   { \__hook_deprecated_generic_warn:w #1 \s__hook_mark }
445 \cs_new_protected:Npn \__hook_deprecated_generic_warn:w
446     #1 / #2 / #3 \s__hook_mark
447   {
448     \if_cs_exist:w __hook~#1/#2/#3 \cs_end: \else:
449       \msg_warning:nnnnn { hooks } { generic-deprecated } {#1} {#2} {#3}
450     \fi:
451     \cs_gset_eq:cN { __hook~#1/#2/#3 } \scan_stop:
452   }
```

`\__hook_do_deprecated_generic:Nn`
`\__hook_do_deprecated_generic:Nw`
`\__hook_declare_deprecated_generic:NNw`
`\__hook_declare_deprecated_generic:NNw`

Now that the user has been told about the deprecation, we proceed by swapping ⟨*name*⟩ and ⟨*place*⟩ and adding the code to the correct hook.

```
453 \cs_new_protected:Npn \__hook_do_deprecated_generic:Nn #1 #2
454   { \__hook_do_deprecated_generic:Nw #1 #2 \s__hook_mark }
455 \cs_new_protected:Npn \__hook_do_deprecated_generic:Nw #1
456         #2 / #3 / #4 \s__hook_mark
457   { #1 { #2 / #4 / #3 } }
458 \cs_new_protected:Npn \__hook_declare_deprecated_generic:NNn #1 #2 #3
459   { \__hook_declare_deprecated_generic:NNw #1 #2 #3 \s__hook_mark }
460 \cs_new_protected:Npn \__hook_declare_deprecated_generic:NNw #1 #2
461     #3 / #4 / #5 \s__hook_mark
462   {
463     \__hook_try_declaring_generic_hook:wnTF #3 / #5 / #4 / \scan_stop:
464       { #3 / #5 / #4 }
465     #1 #2 { #3 / #5 / #4 }
466   }
467 ⟨latexrelease⟩\EndIncludeInRelease
```

```
468 ⟨latexrelease⟩\IncludeInRelease{2021/06/01}{\__hook_try_declaring_generic_hook:wn}
469 ⟨latexrelease⟩                        {Support~cmd~hooks}
470 ⟨latexrelease⟩\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
471 ⟨latexrelease⟩    #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
472 ⟨latexrelease⟩  {
473 ⟨latexrelease⟩    \tl_if_empty:nTF {#2}
474 ⟨latexrelease⟩      { \prg_return_false: }
475 ⟨latexrelease⟩      {
476 ⟨latexrelease⟩        \prop_if_in:NnTF \c__hook_generics_prop {#1}
477 ⟨latexrelease⟩          {
478 ⟨latexrelease⟩            \__hook_if_usable:nF {#5}
479 ⟨latexrelease⟩              {
480 ⟨latexrelease⟩                \str_if_eq:nnT {#1} { cmd }
481 ⟨latexrelease⟩                  { \__hook_try_put_cmd_hook:n {#5} }
482 ⟨latexrelease⟩                \__hook_make_usable:n {#5}
483 ⟨latexrelease⟩              }
484 ⟨latexrelease⟩            \prop_if_in:NnTF \c__hook_generics_reversed_ii_prop {#2}
485 ⟨latexrelease⟩              { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
486 ⟨latexrelease⟩              {
487 ⟨latexrelease⟩                \prop_if_in:NnT \c__hook_generics_reversed_iii_prop {#3}
488 ⟨latexrelease⟩                  { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
489 ⟨latexrelease⟩              }
490 ⟨latexrelease⟩            \prg_return_true:
491 ⟨latexrelease⟩          }
492 ⟨latexrelease⟩          { \prg_return_false: }
493 ⟨latexrelease⟩      }
494 ⟨latexrelease⟩  }
495 ⟨latexrelease⟩\EndIncludeInRelease
496 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_try_declaring_generic_hook:wn}%
497 ⟨latexrelease⟩                        {Support~cmd~hooks}
498 ⟨latexrelease⟩\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
499 ⟨latexrelease⟩    #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
500 ⟨latexrelease⟩  {
501 ⟨latexrelease⟩    \tl_if_empty:nTF {#2}
502 ⟨latexrelease⟩      { \prg_return_false: }
503 ⟨latexrelease⟩      {
504 ⟨latexrelease⟩        \prop_if_in:NnTF \c__hook_generics_prop {#1}
505 ⟨latexrelease⟩          {
506 ⟨latexrelease⟩            \__hook_if_declared:nF {#5} { \hook_new:n {#5} }
507 ⟨latexrelease⟩            \prop_if_in:NnTF \c__hook_generics_reversed_ii_prop {#2}
508 ⟨latexrelease⟩              { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
509 ⟨latexrelease⟩              {
510 ⟨latexrelease⟩                \prop_if_in:NnT \c__hook_generics_reversed_iii_prop {#3}
511 ⟨latexrelease⟩                  { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
512 ⟨latexrelease⟩              }
513 ⟨latexrelease⟩            \prg_return_true:
514 ⟨latexrelease⟩          }
515 ⟨latexrelease⟩          { \prg_return_false: }
516 ⟨latexrelease⟩      }
517 ⟨latexrelease⟩  }
518 ⟨latexrelease⟩
519 ⟨latexrelease⟩\EndIncludeInRelease
```

(*End definition for* __hook_try_declaring_generic_hook:wnTF *and others.*)

`\__hook_if_file_hook_p:w`
`\__hook_if_file_hook:w`_TF_

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

```
520 ⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_if_file_hook:w}%
521 ⟨latexrelease⟩                    {Standardise~generic~hook~names}
522 \prg_new_conditional:Npnn \__hook_if_file_hook:w
523     #1 / #2 \s__hook_mark #3 { TF }
524   {
525     \__hook_if_generic:nTF {#3}
526       {
527         \str_if_eq:nnTF {#1} { file }
528           { \prg_return_true: }
529           { \prg_return_false: }
530       }
531       { \prg_return_false: }
532   }
533 ⟨latexrelease⟩\EndIncludeInRelease
534 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_if_file_hook:w}%
535 ⟨latexrelease⟩                    {Standardise~generic~hook~names}
536 ⟨latexrelease⟩\prg_new_conditional:Npnn \__hook_if_file_hook:w
537 ⟨latexrelease⟩     #1 / #2 / #3 \s__hook_mark { TF }
538 ⟨latexrelease⟩  {
539 ⟨latexrelease⟩     \str_if_eq:nnTF {#1} { file }
540 ⟨latexrelease⟩       {
541 ⟨latexrelease⟩         \bool_lazy_or:nnTF
542 ⟨latexrelease⟩           { \tl_if_empty_p:n {#3} }
543 ⟨latexrelease⟩           { \str_if_eq_p:nn {#3} { / } }
544 ⟨latexrelease⟩           { \prg_return_false: }
545 ⟨latexrelease⟩           {
546 ⟨latexrelease⟩             \prop_if_in:NnTF \c__hook_generics_file_prop {#2}
547 ⟨latexrelease⟩               { \prg_return_true: }
548 ⟨latexrelease⟩               { \prg_return_false: }
549 ⟨latexrelease⟩           }
550 ⟨latexrelease⟩       }
551 ⟨latexrelease⟩       { \prg_return_false: }
552 ⟨latexrelease⟩  }
553 ⟨latexrelease⟩\EndIncludeInRelease
```

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

`\__hook_file_hook_normalize:n`
`\__hook_strip_double_slash:n`
`\__hook_strip_double_slash:w`

```
554 ⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_file_hook_normalize:n}%
555 ⟨latexrelease⟩                    {Standardise~generic~hook~names}
556 ⟨latexrelease⟩\EndIncludeInRelease
```

When a file-specific hook is found, before being declared it is lightly normalized by `\__hook_file_hook_normalize: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/./mypath//file.tex/after}`.

```
557 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_file_hook_normalize:n}%
558 ⟨latexrelease⟩                    {Standardise~generic~hook~names}
559 ⟨latexrelease⟩\cs_new:Npn \__hook_file_hook_normalize:n #1
```

```
560 ⟨latexrelease⟩   { \__hook_strip_double_slash:n {#1} }
561 ⟨latexrelease⟩\cs_new:Npn \__hook_strip_double_slash:n #1
562 ⟨latexrelease⟩   { \__hook_strip_double_slash:w #1 // \s__hook_mark }
```

This function is always called after testing if the argument is a file hook with `\__hook_-`
`if_file_hook:wTF`, so we can assume it has three parts (it is either `file/.../before`
or `file/.../after`), so we use `#1/#2/#3 //` instead of just `#1 //` to prevent losing a
slash if the file name is empty.

```
563 ⟨latexrelease⟩\cs_new:Npn \__hook_strip_double_slash:w #1/#2/#3 // #4 \s__hook_mark
564 ⟨latexrelease⟩   {
565 ⟨latexrelease⟩      \tl_if_empty:nTF {#4}
566 ⟨latexrelease⟩        { #1/#2/#3 }
567 ⟨latexrelease⟩        { \__hook_strip_double_slash:w #1/#2/#3 / #4 \s__hook_mark }
568 ⟨latexrelease⟩   }
569 ⟨latexrelease⟩\EndIncludeInRelease
```

(*End definition for* `\__hook_file_hook_normalize:n`, `\__hook_strip_double_slash:n`, *and* `\__hook_-`
`strip_double_slash:w`.)

`\c__hook_generic_cmd/./before_tl` Token lists defining the possible generic hooks. We don't provide any user interface to
`\c__hook_generic_cmd/./after_tl` this as this is meant to be static.
`\c__hook_generic_env/./before_tl`
`\c__hook_generic_env/./after_tl` **cmd** The generic hooks used for commands.
`\c__hook_generic_file/./before_tl`
`\c__hook_generic_file/./after_tl` **env** The generic hooks used in `\begin` and `\end`.
`\c__hook_generic_package/./before_tl`
`\c__hook_generic_package/./after_tl` **file, package, class, include** The generic hooks used when loading a file
`\c__hook_generic_class/./before_tl`
`\c__hook_generic_class/./after_tl`
`\c__hook_generic_include/./before_tl`
`\c__hook_generic_include/./after_tl`
`\c__hook_generic_env/./begin_tl`
`\c__hook_generic_env/./end_tl`
`\c__hook_generic_include/./end_tl`

```
570 ⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\c__hook_generics_prop}%
571 ⟨latexrelease⟩                          {Standardise~generic~hook~names}
572 \clist_map_inline:nn { cmd , env , file , package , class , include }
573   {
574     \tl_const:cn { c__hook_generic_#1/./before_tl } { + }
575     \tl_const:cn { c__hook_generic_#1/./after_tl  } { - }
576   }
577 \tl_const:cn { c__hook_generic_env/./begin_tl } { + }
578 \tl_const:cn { c__hook_generic_env/./end_tl    } { + }
579 \tl_const:cn { c__hook_generic_include/./end_tl } { - }
```

Deprecated generic hooks:

```
580 \clist_map_inline:nn { file , package , class , include }
581   {
582     \tl_const:cn { c__hook_deprecated_#1/./before_tl } { }
583     \tl_const:cn { c__hook_deprecated_#1/./after_tl  } { }
584   }
585 \tl_const:cn { c__hook_deprecated_include/./end_tl } { }
586 ⟨latexrelease⟩\EndIncludeInRelease

587 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\c__hook_generics_prop}%
588 ⟨latexrelease⟩                          {Standardise~generic~hook~names}
589 ⟨latexrelease⟩\prop_const_from_keyval:Nn \c__hook_generics_prop
590 ⟨latexrelease⟩    {cmd=,env=,file=,package=,class=,include=}
591 ⟨latexrelease⟩\EndIncludeInRelease
```

(*End definition for* `\c__hook_generic_cmd/./before_tl` *and others.*)

The following generic hooks are supposed to use reverse ordering (the `ii` and `iii` names are kept for the deprecation cycle):

`\c_hook_generics_reversed_ii_prop`
`\c_hook_generics_reversed_iii_prop`
`\c__hook_generics_file_prop`

```
592 ⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\c__hook_generics_reversed_ii_prop}%
593 ⟨latexrelease⟩                    {Standardise~generic~hook~names}
594 ⟨latexrelease⟩\EndIncludeInRelease

595 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\c__hook_generics_reversed_ii_prop}%
596 ⟨latexrelease⟩                    {Standardise~generic~hook~names}
597 ⟨latexrelease⟩\prop_const_from_keyval:Nn \c__hook_generics_reversed_ii_prop {after=,end=}
598 ⟨latexrelease⟩\prop_const_from_keyval:Nn \c__hook_generics_reversed_iii_prop {after=}
599 ⟨latexrelease⟩\prop_const_from_keyval:Nn \c__hook_generics_file_prop {before=,after=}
600 ⟨latexrelease⟩\EndIncludeInRelease
```

(*End definition for* `\c_hook_generics_reversed_ii_prop`, `\c_hook_generics_reversed_iii_prop`, *and* `\c__hook_generics_file_prop`.)

`\hook_gremove_code:nn`
`\__hook_gremove_code:nn`

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

```
601 \cs_new_protected:Npn \hook_gremove_code:nn #1 #2
602   { \__hook_normalize_hook_args:Nnn \__hook_gremove_code:nn {#1} {#2} }
603 \cs_new_protected:Npn \__hook_gremove_code:nn #1 #2
604   {
```

First check that the hook code pool exists. `\__hook_if_usable:nTF` isn't used here because it should be possible to remove code from a hook before its defined (see section 2.1.8).

```
605     \__hook_if_structure_exist:nTF {#1}
606       {
```

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}`.

If all code is to be removed, clear the code pool `\g__hook_⟨hook⟩_code_prop`, the top-level code `\__hook_toplevel␣⟨hook⟩`, and the next-execution code `\__hook_-next␣⟨hook⟩`.

```
607         \str_if_eq:nnTF {#2} {*}
608           {
609             \prop_gclear:c { g__hook_#1_code_prop }
610             \__hook_tl_gclear:c { __hook_toplevel~#1 }
611             \__hook_tl_gclear:c { __hook_next~#1 }
612           }
613           {
```

If the label is `top-level` then clear the token list, as all code there is under the same label.

```
614             \str_if_eq:nnTF {#2} { top-level }
615               { \__hook_tl_gclear:c { __hook_toplevel~#1 } }
616               {
617                 \prop_gpop:cnNF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
618                   { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
619               }
620           }
```

Finally update the code, if the hook exists.

```
621         \__hook_if_usable:nT {#1}
622           { \__hook_update_hook_code:n {#1} }
623       }
```

If the code pool for this hook doesn't exist, show a warning:

```
624     {
625       \__hook_if_deprecated_generic:nTF {#1}
626         {
627           \__hook_deprecated_generic_warn:n {#1}
628           \__hook_do_deprecated_generic:Nn \__hook_gremove_code:nn {#1} {#2}
629         }
630         { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
631     }
632   }
```

(*End definition for* `\hook_gremove_code:nn` *and* `\__hook_gremove_code:nn`. *This function is documented on page 14.*)

`\g__hook_??_code_prop`
`\__hook~??`
`\g__hook_??_reversed_tl`

Initially these variables 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, the code should be fast, so this is not done with `c` expansion in the code later on.

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

```
633 \prop_new:c {g__hook_??_code_prop}
634 \prop_new:c {__hook~??}
```

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.

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

(*End definition for* `\g__hook_??_code_prop`, `\__hook~??`, *and* `\g__hook_??_reversed_tl`.)

4.7   Setting rules for hooks code

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

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, which sets a default rule (to be used if no other relation between the two hooks exist).

```
636 \cs_new_protected:Npn \hook_gset_rule:nnnn #1#2#3#4
637   {
638     \__hook_normalize_hook_rule_args:Nnnnn \__hook_gset_rule:nnnn
639       {#1} {#2} {#3} {#4}
640   }
641 \cs_new_protected:Npn \__hook_gset_rule:nnnn #1#2#3#4
642   {
643     \__hook_if_deprecated_generic:nT {#1}
644       {
645         \__hook_deprecated_generic_warn:n {#1}
646         \__hook_do_deprecated_generic:Nn \__hook_gset_rule:nnnn {#1}
647           {#2} {#3} {#4}
648         \exp_after:wN \use_none:nnnnnnnn \use_none:n
649       }
```

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

```
650        \__hook_init_structure:n {#1}
```

Then we clear any previous relationship between both labels.

```
651        \__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.

```
652        \cs_if_exist_use:cTF { __hook_rule_#3_gset:nnn }
653          {
654             {#1} {#2} {#4}
655             \__hook_update_hook_code:n {#1}
656          }
657          { \msg_error:nnnnnn { hooks } { unknown-rule }
658                               {#1} {#2} {#3} {#4}          }
659     }
```

(*End definition for* `\hook_gset_rule:nnnn` *and* `\__hook_gset_rule:nnnn`. *This function is documented on page* *14*.)

`\__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 $l_B < l_A$ only presented differently. But by normalizing the forms of the rule to a single representation, say, $l_B < l_A$, reduces the time spent looking for the rules later considerably.

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 $l_B | l_A$ 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.

```
660  \cs_new_protected:Npn \__hook_rule_before_gset:nnn #1#2#3
661     {
662        \__hook_tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
663          { \__hook_label_ordered:nnTF {#2} {#3} { < } { > } }
664     }
665  \cs_new_eq:cN { __hook_rule_<_gset:nnn } \__hook_rule_before_gset:nnn

666  \cs_new_protected:Npn \__hook_rule_after_gset:nnn #1#2#3
667     {
668        \__hook_tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#3} {#2} _tl }
669          { \__hook_label_ordered:nnTF {#3} {#2} { < } { > } }
670     }
671  \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_voids_gset:nnn`

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

```
672  \cs_new_protected:Npn \__hook_rule_voids_gset:nnn #1#2#3
673     {
674        \__hook_tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
675          { \__hook_label_ordered:nnTF {#2} {#3} { -> } { <- } }
676     }
```

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

48

These relations make an error/warning if labels #2 and #3 appear together in hook #1.

`\_hook_rule_incompatible-error_gset:nnn`
`\_hook_rule_incompatible-warning_gset:nnn`

```
677 \cs_new_protected:cpn { __hook_rule_incompatible-error_gset:nnn } #1#2#3
678   { \__hook_tl_gset:cn { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
679             { xE } }
680 \cs_new_protected:cpn { __hook_rule_incompatible-warning_gset:nnn } #1#2#3
681   { \__hook_tl_gset:cn { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
682             { 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.

```
683 \cs_new_protected:Npn \__hook_rule_unrelated_gset:nnn #1#2#3 { }
684 \cs_new_protected:Npn \__hook_rule_gclear:nnn #1#2#3
685   { \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.

```
686 \cs_new:Npn \__hook_label_pair:nn #1#2
687   {
688     \if_case:w \__hook_str_compare:nn {#1} {#2} \exp_stop_f:
689             #1 | #1 %  0
690     \or:    #1 | #2 % +1
691     \else: #2 | #1 % -1
692     \fi:
693   }
```

(*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.

```
694 \prg_new_conditional:Npnn \__hook_label_ordered:nn #1#2 { TF }
695   {
696     \if_int_compare:w \__hook_str_compare:nn {#1} {#2} > 0 \exp_stop_f:
697       \prg_return_true:
698     \else:
699       \prg_return_false:
700     \fi:
701   }
```

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

`\__hook_if_label_case:nnnnn`

To avoid doing the string comparison twice in `\__hook_initialize_single:NNn` (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.

```
702 \cs_new:Npn \__hook_if_label_case:nnnnn #1#2
703   {
704     \cs:w use_
705       \if_case:w \__hook_str_compare:nn {#1} {#2}
706         i \or: ii \else: iii \fi: :nnn
707     \cs_end:
708   }
```

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

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

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

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

`\__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.

710 `\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.

711 `    \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.

712 `    \__hook_debug:n { \prop_gclear:N \g__hook_used_prop }`
713 `    \seq_map_inline:Nn \g__hook_all_seq`
714 `        {`
715 `            \__hook_update_hook_code:n {##1}`
716 `        }`

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

717 `    \__hook_debug:n`
718 `        { \iow_term:x{^^JAll~ initialized~ (non-empty)~ hooks:}`
719 `            \prop_map_inline:Nn \g__hook_used_prop`
720 `                { \iow_term:x{^^J~ ##1~ ->~`
721 `                    \exp_not:v {__hook~##1}~ }`
722 `                }`
723 `        }`

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.

724 `    \cs_gset_eq:NN \hook_use:n \__hook_use_initialized:n`
725 `    \cs_gset_eq:NN \__hook_preamble_hook:n \use_none:n`
726 `}`

(*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.

727 `\cs_new_protected:Npn \__hook_initialize_hook_code:n #1`
728 `    {`
729 `        \__hook_debug:n{ \iow_term:x{^^JUpdate~ code~ for~ hook~`
730 `                                        '#1' \on@line :^^J} }`

This does the sorting and the updates. First thing we do is to check if a legacy hook macro exists and if so we add it to the hook under the label `legacy`. This might make the hook non-empty so we have to do this before the then following test.

731 `        \__hook_include_legacy_code_chunk:n {#1}`

50

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 __-hook␣⟨hook⟩ to hold the top-level and next code chunks. If there are code chunks we call __hook_initialize_single:NNn 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.

```
732        \__hook_if_usable:nT {#1}
733          {
734            \prop_if_empty:cTF { g__hook_#1_code_prop }
735              {
736                \__hook_tl_gset:co { __hook~#1 }
737                  {
738                    \cs:w __hook_toplevel~#1 \exp_after:wN \cs_end:
739                    \cs:w __hook_next~#1 \cs_end:
740                  }
741              }
742              {
```

By default the algorithm sorts the code chunks and then saves the result in a token list for fast execution; this is done by adding the code chunks 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 that are used later on.

```
743                \__hook_if_reversed:nTF {#1}
744                  { \cs_set_eq:NN \__hook_tl_gput:Nn    \__hook_tl_gput_left:Nn
745                    \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_left:NV  }
746                  { \cs_set_eq:NN \__hook_tl_gput:Nn    \__hook_tl_gput_right:Nn
747                    \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_right:NV }
```

When sorting, some relations (namely voids) need to act destructively on the code property lists to remove code that shouldn't appear in the sorted hook token list, so we make a copy of the code property list that we can safely work on without changing the main one.

```
748                \prop_set_eq:Nc \l__hook_work_prop { g__hook_#1_code_prop }
749                \__hook_initialize_single:ccn
750                  { __hook~#1 } { g__hook_#1_labels_clist } {#1}
```

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.

```
751                \__hook_debug:n{ \exp_args:NNx \prop_gput:Nnn
752                                      \g__hook_used_prop {#1}{} }
753              }
754          }
755      }
```

(*End definition for* __hook_initialize_hook_code:n.)

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

>   *FMi: note to myself: verify*

```
756 \cs_new:Npn \__hook_tl_csname:n #1 { l__hook_label_#1_tl }
757 \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:

<div style="margin-left:2em">

758   `\seq_new:N \l__hook_labels_seq`

</div>

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

<div style="margin-left:2em">

759   `\int_new:N \l__hook_labels_int`

</div>

- 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*:

<div style="margin-left:2em">

760   `\tl_new:N \l__hook_front_tl`
761   `\tl_new:N \l__hook_rear_tl`

</div>

- 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:

<div style="margin-left:2em">

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

</div>

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

`\__hook_initialize_single:NNn`
`\__hook_initialize_single:ccn`

`\__hook_initialize_single:NNn` implements the sorting of the code chunks for a hook and saves the result in the token list for fast execution (#4). The arguments are ⟨*hook-code-plist*⟩, ⟨*hook-code-tl*⟩, ⟨*hook-top-level-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 … maybe

763 `\cs_new_protected:Npn \__hook_initialize_single:NNn #1#2#3`
764   `{`

Step T1: Initialize the data structure …

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

Store the name of the hook:

767   `\tl_set:Nn \l__hook_cur_hook_tl {#3}`

We loop over the property list holding the code and record all the labels listed there. Only the 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.

```
768      \prop_map_inline:Nn \l__hook_work_prop
769        {
770          \int_incr:N \l__hook_labels_int
771          \seq_put_right:Nn \l__hook_labels_seq {##1}
772          \__hook_tl_set:cn { \__hook_tl_csname:n {##1} } { 0 }
773          \seq_clear_new:c { \__hook_seq_csname:n {##1} }
774        }
```

Steps T2 and T3: Here we sort the relevant rules into the data structure. . .

This loop constitutes a square matrix of the labels in `\l__hook_work_prop` 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.

```
775      \prop_map_inline:Nn \l__hook_work_prop
776        {
777          \prop_map_inline:Nn \l__hook_work_prop
778            {
779              \__hook_if_label_case:nnnnn {##1} {####1}
780                { \prop_map_break: }
781                { \__hook_apply_label_pair:nnn {##1} {####1} }
782                { \__hook_apply_label_pair:nnn {####1} {##1} }
783                  {#3}
784            }
785        }
```

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

```
786      \__hook_debug:n { \__hook_debug_label_data:N \l__hook_work_prop }
```

Step T4:

```
787      \tl_set:Nn \l__hook_rear_tl { 0 }
788      \tl_set:cn { \__hook_tl_csname:n { 0 } } { 0 }
789      \seq_map_inline:Nn \l__hook_labels_seq
790        {
791          \int_compare:nNnT { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
792            {
793              \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } }{##1}
794              \tl_set:Nn \l__hook_rear_tl {##1}
795            }
796        }
797      \tl_set_eq:Nc \l__hook_front_tl { \__hook_tl_csname:n { 0 } }
798      \__hook_tl_gclear:N #1
799      \clist_gclear:N #2
```

The whole loop gets combined in steps T5–T7:

```
800      \bool_while_do:nn { ! \str_if_eq_p:Vn \l__hook_front_tl { 0 } }
801        {
```

This part is step T5:

```
802          \int_decr:N \l__hook_labels_int
803          \prop_get:NVN \l__hook_work_prop \l__hook_front_tl \l__hook_return_tl
804          \exp_args:NNV \__hook_tl_gput:Nn #1 \l__hook_return_tl

805          \__hook_clist_gput:NV #2 \l__hook_front_tl
806          \__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.

```
807          \seq_map_inline:cn { \__hook_seq_csname:n { \l__hook_front_tl } }
808            {
809              \tl_set:cx { \__hook_tl_csname:n {##1} }
810                        { \int_eval:n
811                            { \cs:w \__hook_tl_csname:n {##1} \cs_end: - 1 }
812                        }
813              \int_compare:nNnT
814                  { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
815                  {
816                    \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } } {##1}
817                    \tl_set:Nn \l__hook_rear_tl            {##1}
818                  }
819            }
```

and here is step T7:

```
820          \tl_set_eq:Nc \l__hook_front_tl
821                        { \__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.

```
822        }
823      \int_compare:nNnF \l__hook_labels_int = 0
824        {
825          \iow_term:x{====================}
826          \iow_term:x{Error:~ label~ rules~ are~ incompatible:}
```

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

FMi: improve output on a rainy day

```
827          \__hook_debug_label_data:N \l__hook_work_prop
828          \iow_term:x{====================}
829        }
```

After we have added all hook code to `#1`, we finish it off by adding extra code for the `top-level` (`#2`) and for one time execution (`#3`). These should normally be empty. The `top-level` code is added with `\__hook_tl_gput:Nn` as that might change for a reversed hook (then `top-level` is the very first code chunk added). The `next` code is always added last.

```
830      \exp_args:NNo \__hook_tl_gput:Nn #1 { \cs:w __hook_toplevel~#3 \cs_end: }
831      \__hook_tl_gput_right:No #1 { \cs:w __hook_next~#3 \cs_end: }
832    }

833 \cs_generate_variant:Nn \__hook_initialize_single:NNn { cc }
```

(*End definition for* __hook_initialize_single:NNn.)

__hook_tl_gput:Nn
__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.

```
834 \cs_new:Npn \__hook_tl_gput:Nn   { \ERROR }
835 \cs_new:Npn \__hook_clist_gput:NV { \ERROR }
```

(*End definition for* __hook_tl_gput:Nn *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.

```
836 \cs_new_protected:Npn \__hook_apply_label_pair:nnn #1#2#3
837   {
```

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.

```
838     \__hook_label_if_exist_apply:nnnF {#1} {#2} {#3}
839       {
```

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

```
840         \__hook_label_if_exist_apply:nnnF {#1} {#2} { ?? } { }
841       }
842   }
843 \cs_new_protected:Npn \__hook_label_if_exist_apply:nnnF #1#2#3
844   {
845     \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.

```
846         \__hook_apply_rule:nnn {#1} {#2} {#3}
847         \exp_after:wN \use_none:n
848     \else:
849         \use:nn
850     \fi:
851   }
```

(*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⟩.

```
852 \cs_new_protected:Npn \__hook_apply_rule:nnn #1#2#3
853   {
854     \cs:w __hook_apply_
```

```
855        \cs:w g__hook_#3_reversed_tl \cs_end: rule_
856          \cs:w g__hook_ #3 _rule_ #1 | #2 _tl \cs_end: :nnn \cs_end:
857        {#1} {#2} {#3}
858      }
```

(*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.

```
859 \cs_new_protected:cpn { __hook_apply_rule_<:nnn } #1#2#3
860    {
861      \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
862      \tl_set:cx { \__hook_tl_csname:n {#2} }
863        { \int_eval:n{ \cs:w \__hook_tl_csname:n {#2} \cs_end: + 1 } }
864      \seq_put_right:cn{ \__hook_seq_csname:n {#1} }{#2}
865    }
866 \cs_new_protected:cpn { __hook_apply_rule_>:nnn } #1#2#3
867    {
868      \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
869      \tl_set:cx { \__hook_tl_csname:n {#1} }
870        { \int_eval:n{ \cs:w \__hook_tl_csname:n {#1} \cs_end: + 1 } }
871      \seq_put_right:cn{ \__hook_seq_csname:n {#2} }{#1}
872    }
```

(*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.

```
873 \cs_new_protected:cpn { __hook_apply_rule_xE:nnn } #1#2#3
874    {
875      \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
876      \msg_error:nnnnnn { hooks } { labels-incompatible }
877        {#1} {#2} {#3} { 1 }
878      \use:c { __hook_apply_rule_->:nnn } {#1} {#2} {#3}
879      \use:c { __hook_apply_rule_<-:nnn } {#1} {#2} {#3}
880    }
881 \cs_new_protected:cpn { __hook_apply_rule_xW:nnn } #1#2#3
882    {
883      \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
884      \msg_warning:nnnnnn { hooks } { labels-incompatible }
885        {#1} {#2} {#3} { 0 }
886    }
```

(*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 drop everything for that label since it doesn't matter where we put such empty code. However that would complicate the algorithm a lot with little gain.[8] 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.

---

[8]This also has the advantage that the result of the sorting doesn't change, as it might otherwise do (for unrelated chunks) if we aren't careful.

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 it instead from `\l__hook_cur_hook_tl` makes the default rules `->` and `<-` work properly.

```
887 \cs_new_protected:cpn { __hook_apply_rule_->:nnn } #1#2#3
888   {
889     \__hook_debug:n
890       {
891         \__hook_msg_pair_found:nnn {#1} {#2} {#3}
892         \iow_term:x{--->~ Drop~ '#2'~ code~ from~
893           \iow_char:N \\ g__hook_ \l__hook_cur_hook_tl _code_prop ~
894           because~ of~ '#1' }
895       }
896     \prop_put:Nnn \l__hook_work_prop {#2} { }
897   }
898 \cs_new_protected:cpn { __hook_apply_rule_<-:nnn } #1#2#3
899   {
900     \__hook_debug:n
901       {
902         \__hook_msg_pair_found:nnn {#1} {#2} {#3}
903         \iow_term:x{--->~ Drop~ '#1'~ code~ from~
904           \iow_char:N \\ g__hook_ \l__hook_cur_hook_tl _code_prop ~
905           because~ of~ '#2' }
906       }
907     \prop_put:Nnn \l__hook_work_prop {#1} { }
908   }
```

(*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.

```
909 \cs_new_eq:cc { __hook_apply_-rule_<:nnn  } { __hook_apply_rule_>:nnn }
910 \cs_new_eq:cc { __hook_apply_-rule_>:nnn  } { __hook_apply_rule_<:nnn }
911 \cs_new_eq:cc { __hook_apply_-rule_<-:nnn } { __hook_apply_rule_<-:nnn }
912 \cs_new_eq:cc { __hook_apply_-rule_->:nnn } { __hook_apply_rule_->:nnn }
913 \cs_new_eq:cc { __hook_apply_-rule_xE:nnn  } { __hook_apply_rule_xE:nnn }
914 \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.

```
915 \cs_new_protected:Npn \__hook_msg_pair_found:nnn #1#2#3
916   {
917     \iow_term:x{~ \str_if_eq:nnTF {#3} {??} {default} {~normal} ~
918         rule~ \__hook_label_pair:nn {#1} {#2}:~
919         \use:c { g__hook_#3_rule_ \__hook_label_pair:nn {#1} {#2} _tl } ~
920         found}
921   }
```

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

`\__hook_debug_label_data:N`

```
922 \cs_new_protected:Npn \__hook_debug_label_data:N #1 {
923   \iow_term:x{Code~ labels~ for~ sorting:}
924   \iow_term:x{~ \seq_use:Nnnn\l__hook_labels_seq {~and~}{,~}{~and~} }
925   \iow_term:x{^^J Data~ structure~ for~ label~ rules:}
```

```
926    \prop_map_inline:Nn #1
927      {
928        \iow_term:x{~ ##1~ =~ \tl_use:c{ \__hook_tl_csname:n {##1} }~ ->~
929          \seq_use:cnnn{ \__hook_seq_csname:n {##1} }{~->~}{~->~}{~->~}
930      }
931    }
932    \iow_term:x{}
933 }
```

(*End definition for* __hook_debug_label_data:N.)

\hook_show:n   This writes out information about the hook given in its argument onto the .log file and
\hook_log:n   the terminal, if \show_hook:n is used. Internally both share the same structure, except
__hook_log_line:x   that at the end, \hook_show:n triggers TeX's prompt.
__hook_log_line_indent:x
__hook_log:nN

```
934 \cs_new_protected:Npn \hook_log:n #1
935   {
936     \cs_set_eq:NN \__hook_log_cmd:x \iow_log:x
937     \__hook_normalize_hook_args:Nn \__hook_log:nN {#1} \tl_log:x
938   }
939 \cs_new_protected:Npn \hook_show:n #1
940   {
941     \cs_set_eq:NN \__hook_log_cmd:x \iow_term:x
942     \__hook_normalize_hook_args:Nn \__hook_log:nN {#1} \tl_show:x
943   }
944 \cs_new_protected:Npn \__hook_log_line:x #1
945   { \__hook_log_cmd:x { >~#1 } }
946 \cs_new_protected:Npn \__hook_log_line_indent:x #1
947   { \__hook_log_cmd:x { >~\@spaces #1 } }
948 \cs_new_protected:Npn \__hook_log:nN #1 #2
949   {
950     \__hook_if_deprecated_generic:nT {#1}
951       {
952         \__hook_deprecated_generic_warn:n {#1}
953         \__hook_do_deprecated_generic:Nn \__hook_log:nN {#1} #2
954         \exp_after:wN \use_none:nnnnnnnnn \use_none:nnnnn
955       }
956     \__hook_preamble_hook:n {#1}
957     \__hook_log_cmd:x
958       { ^^J ->~The~ \__hook_if_generic:nT {#1} { generic~ } hook~'#1': }
959     \__hook_if_usable:nF {#1}
960       { \__hook_log_line:x { The~hook~is~not~declared. } }
961     \__hook_if_disabled:nT {#1}
962       { \__hook_log_line:x { The~hook~is~disabled. } }
963     \hook_if_empty:nTF {#1}
964       { #2 { The~hook~is~empty } }
965       {
966         \__hook_log_line:x { Code~chunks: }
967         \prop_if_empty:cTF { g__hook_#1_code_prop }
968           { \__hook_log_line_indent:x { --- } }
969           {
970             \prop_map_inline:cn { g__hook_#1_code_prop }
971               { \__hook_log_line_indent:x { ##1~->~\tl_to_str:n {##2} } }
972           }
```

58

If there is code in the `top-level` token list, print it:

```
973          \__hook_log_line:x
974            {
975              Document-level~(top-level)~code
976              \__hook_if_usable:nT {#1}
977                { ~(executed~\__hook_if_reversed:nTF {#1} {first} {last} ) } :
978            }
979          \__hook_log_line_indent:x
980            {
981              \tl_if_empty:cTF { __hook_toplevel~#1 }
982                { --- }
983                { -> ~ \exp_args:Nv \tl_to_str:n { __hook_toplevel~#1 } }
984            }
985          \__hook_log_line:x { Extra~code~for~next~invocation: }
986          \__hook_log_line_indent:x
987            {
988              \tl_if_empty:cTF { __hook_next~#1 }
989                { --- }
```

If the token list is not empty we want to display it but without the first tokens (the code to clear itself) so we call a helper command to get rid of them.

```
990                { ->~ \exp_args:Nv \__hook_log_next_code:n { __hook_next~#1 } }
991            }
```

Loop through the rules in a hook and for every rule found, print it. If no rule is there, print `---`. The boolean `\l__hook_tmpa_bool` here indicates if the hook has no rules.

```
992          \__hook_log_line:x { Rules: }
993          \bool_set_true:N \l__hook_tmpa_bool
994          \__hook_list_rules:nn {#1}
995            {
996              \bool_set_false:N \l__hook_tmpa_bool
997              \__hook_log_line_indent:x
998                {
999                  ##2~ with~
1000                 \str_if_eq:nnT {##3} {??} { default~ }
1001                 relation~ ##1
1002               }
1003           }
1004         \bool_if:NT \l__hook_tmpa_bool
1005           { \__hook_log_line_indent:x { --- } }
```

When the hook is declared (that is, the sorting algorithm is applied to that hook) and not empty

```
1006         \bool_lazy_and:nnTF
1007             { \__hook_if_usable_p:n {#1} }
1008             { ! \hook_if_empty_p:n {#1} }
1009           {
1010             \__hook_log_line:x
1011               {
1012                 Execution~order
1013                 \bool_if:NTF \l__hook_tmpa_bool
1014                   { \__hook_if_reversed:nT {#1} { ~(after~reversal) } }
1015                   { ~(after~
```

```
1016                        \__hook_if_reversed:nT {#1} { reversal~and~ }
1017                        applying~rules)
1018                      } :
1019                    }
1020                  #2 % \tl_show:n
1021                    {
1022                      \@spaces
1023                      \clist_if_empty:cTF { g__hook_#1_labels_clist }
1024                        { --- }
1025                        { \clist_use:cn {g__hook_#1_labels_clist} { ,~ } }
1026                    }
1027              }
1028              {
1029                \__hook_log_line:x { Execution~order: }
1030                #2
1031                  {
1032                    \@spaces Not~set~because~the~hook~ \__hook_if_usable:nTF {#1}
1033                      { code~pool~is~empty }
1034                      { is~\__hook_if_disabled:nTF {#1} {disabled} {undeclared} }
1035                  }
1036              }
1037          }
1038      }
```

To display the code for next invocation only (i.e., from `\AddToHookNext` we have to remove the first two tokens at the front which are `\tl_gclear:N` and the token list to clear.

```
1039  \cs_new:Npn \__hook_log_next_code:n #1
1040    { \exp_args:No \tl_to_str:n { \use_none:nn #1 } }
```

__hook_log_next_code:n

(*End definition for* `\hook_show:n` *and others. These functions are documented on page* *14.*)

__hook_list_rules:nn
__hook_list_one_rule:nnn
__hook_list_if_rule_exists:nnnF

This macro takes a ⟨*hook*⟩ and an ⟨*inline function*⟩ and loops through each pair of ⟨*labels*⟩ in the ⟨*hook*⟩, and if there is a relation between this pair of ⟨*labels*⟩, the ⟨*inline function*⟩ is executed with #1 = ⟨*relation*⟩, #2 = ⟨*label₁*⟩|⟨*label₂*⟩, and #3 = ⟨*hook*⟩ (the latter may be the argument #1 to `\__hook_list_rules:nn`, or ?? if it is a default rule).

```
1041  \cs_new_protected:Npn \__hook_list_rules:nn #1 #2
1042    {
1043      \cs_set_protected:Npn \__hook_tmp:w ##1 ##2 ##3 {#2}
1044      \prop_map_inline:cn { g__hook_#1_code_prop }
1045        {
1046          \prop_map_inline:cn { g__hook_#1_code_prop }
1047            {
1048              \__hook_if_label_case:nnnnn {##1} {####1}
1049                { \prop_map_break: }
1050                { \__hook_list_one_rule:nnn {##1} {####1} }
1051                { \__hook_list_one_rule:nnn {####1} {##1} }
1052                  {#1}
1053            }
1054        }
1055    }
```

These two are quite similar to `\__hook_apply_label_pair:nnn` and `\__hook_-label_if_exist_apply:nnnF`, respectively, but rather than applying the rule, they pass it to the ⟨*inline function*⟩.

```
1056 \cs_new_protected:Npn \__hook_list_one_rule:nnn #1#2#3
1057   {
1058     \__hook_list_if_rule_exists:nnnF {#1} {#2} {#3}
1059       { \__hook_list_if_rule_exists:nnnF {#1} {#2} { ?? } { } }
1060   }
1061 \cs_new_protected:Npn \__hook_list_if_rule_exists:nnnF #1#2#3
1062   {
1063     \if_cs_exist:w g__hook_ #3 _rule_ #1 | #2 _tl \cs_end:
1064       \exp_args:Nv \__hook_tmp:w
1065         { g__hook_ #3 _rule_ #1 | #2 _tl } { #1 | #2 } {#3}
1066       \exp_after:wN \use_none:nn
1067     \fi:
1068     \use:n
1069   }
```

(*End definition for* `\__hook_list_rules:nn`, `\__hook_list_one_rule:nnn`, *and* `\__hook_list_if_-rule_exists:nnnF`.)

`\__hook_debug_print_rules:n` A shorthand for debugging that prints similar to `\prop_show:N`.

```
1070 \cs_new_protected:Npn \__hook_debug_print_rules:n #1
1071   {
1072     \iow_term:n { The~hook~#1~contains~the~rules: }
1073     \cs_set_protected:Npn \__hook_tmp:w ##1
1074       {
1075         \__hook_list_rules:nn {#1}
1076           {
1077             \iow_term:x
1078               {
1079                 > ##1 {####2} ##1 => ##1 {####1}
1080                 \str_if_eq:nnT {####3} {??} { ~(default) }
1081               }
1082           }
1083       }
1084     \exp_args:No \__hook_tmp:w { \use:nn { ~ } { ~ } }
1085   }
```

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

4.8   Specifying code for next invocation

`\hook_gput_next_code:nn`

```
1086 \cs_new_protected:Npn \hook_gput_next_code:nn #1
1087   { \__hook_normalize_hook_args:Nn \__hook_gput_next_code:nn {#1} }
```

(*End definition for* `\hook_gput_next_code:nn`. *This function is documented on page 13.*)

`\__hook_gput_next_code:nn`
`\__hook_gput_next_do:nn`
`\__hook_gput_next_do:Nnn`

```
1088 \cs_new_protected:Npn \__hook_gput_next_code:nn #1 #2
1089   {
1090     \__hook_if_disabled:nTF {#1}
1091       { \msg_error:nnn { hooks } { hook-disabled } {#1} }
```

```
1092        {
1093          \__hook_if_structure_exist:nTF {#1}
1094            { \__hook_gput_next_do:nn {#1} {#2} }
1095            { \__hook_try_declaring_generic_next_hook:nn {#1} {#2} }
1096        }
1097    }
1098 \cs_new_protected:Npn \__hook_gput_next_do:nn #1
1099    {
1100      \exp_args:Nc \__hook_gput_next_do:Nnn
1101        { __hook_next~#1 } {#1}
1102    }
```

First check if the "next code" token list is empty: if so we need to add a `\tl_gclear:c`
to clear it, so the code lasts for one usage only. The token list is cleared early so that
nested usages don't get lost. `\tl_gclear:c` is used instead of `\tl_gclear:N` in case
the hook is used in an expansion-only context, so the token list doesn't expand before
`\tl_gclear:N`: that would make an infinite loop. Also in case the main code token list
is empty, the hook code has to be updated to add the next execution token list.

```
1103 \cs_new_protected:Npn \__hook_gput_next_do:Nnn #1 #2
1104    {
1105      \tl_if_empty:cT { __hook~#2 }
1106        { \__hook_update_hook_code:n {#2} }
1107      \tl_if_empty:NT #1
1108        { \__hook_tl_gset:Nn #1 { \__hook_clear_next:n {#2} } }
1109      \__hook_tl_gput_right:Nn #1
1110    }
```

(*End definition for* `\__hook_gput_next_code:nn`, `\__hook_gput_next_do:nn`, *and* `\__hook_gput_next_-`
`do:Nnn`.)

<div style="float:left">

`\hook_gclear_next_code:n`
`\__hook_clear_next:n`

</div>

Discard anything set up for next invocation of the hook.

```
1111 \cs_new_protected:Npn \hook_gclear_next_code:n #1
1112    { \__hook_normalize_hook_args:Nn \__hook_clear_next:n {#1} }
1113 \cs_new_protected:Npn \__hook_clear_next:n #1
1114    { \cs_gset_eq:cN { __hook_next~#1 } \c_empty_tl }
```

(*End definition for* `\hook_gclear_next_code:n` *and* `\__hook_clear_next:n`. *This function is documented on page 13.*)

4.9   Using the hook

<div style="float:left">

`\hook_use:n`
`\__hook_use_initialized:n`
`\__hook_use_undefined:w`
`\__hook_use_end:`
`\__hook_preamble_hook:n`

</div>

`\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 thing internally: they check that the hook exists as given,
and if so they use it as quickly as possible.

   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.

```
1115 ⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\hook_use:n}
1116 ⟨latexrelease⟩                          {Standardise~generic~hook~names}
1117 \cs_new_protected:Npn \hook_use:n #1
1118   {
1119     \tl_if_exist:cT { __hook~#1 }
1120       {
1121         \__hook_preamble_hook:n {#1}
1122         \cs:w __hook~#1 \cs_end:
1123       }
1124   }
1125 \cs_new:Npn \__hook_use_initialized:n #1
1126   {
1127     \if_cs_exist:w __hook~#1 \cs_end:
1128       \cs:w __hook~#1 \exp_after:wN \cs_end:
1129     \fi:
1130   }
1131 \cs_new_protected:Npn \__hook_preamble_hook:n #1
1132   { \__hook_initialize_hook_code:n {#1} }
1133 ⟨latexrelease⟩\EndIncludeInRelease

1134 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_use:n}
1135 ⟨latexrelease⟩                          {Standardise~generic~hook~names}
1136 ⟨latexrelease⟩\cs_new_protected:Npn \hook_use:n #1
1137 ⟨latexrelease⟩  {
1138 ⟨latexrelease⟩    \tl_if_exist:cTF { __hook~#1 }
1139 ⟨latexrelease⟩      {
1140 ⟨latexrelease⟩        \__hook_preamble_hook:n {#1}
1141 ⟨latexrelease⟩        \cs:w __hook~#1 \cs_end:
1142 ⟨latexrelease⟩      }
1143 ⟨latexrelease⟩      { \__hook_use:wn #1 / \s__hook_mark {#1} }
1144 ⟨latexrelease⟩  }
1145 ⟨latexrelease⟩\cs_new:Npn \__hook_use_initialized:n #1
1146 ⟨latexrelease⟩  {
1147 ⟨latexrelease⟩    \if_cs_exist:w __hook~#1 \cs_end:
1148 ⟨latexrelease⟩    \else:
1149 ⟨latexrelease⟩      \__hook_use_undefined:w
1150 ⟨latexrelease⟩    \fi:
1151 ⟨latexrelease⟩    \cs:w __hook~#1 \__hook_use_end:
1152 ⟨latexrelease⟩  }
1153 ⟨latexrelease⟩\cs_new:Npn \__hook_use_undefined:w #1 #2 __hook~#3 \__hook_use_end:
1154 ⟨latexrelease⟩  {
1155 ⟨latexrelease⟩    #1 % fi
1156 ⟨latexrelease⟩    \__hook_use:wn #3 / \s__hook_mark {#3}
1157 ⟨latexrelease⟩  }
1158 ⟨latexrelease⟩\cs_new_protected:Npn \__hook_preamble_hook:n #1
1159 ⟨latexrelease⟩  { \__hook_initialize_hook_code:n {#1} }
1160 ⟨latexrelease⟩\cs_new_eq:NN \__hook_use_end: \cs_end:
1161 ⟨latexrelease⟩\EndIncludeInRelease
```

(*End definition for* \hook_use:n *and others. This function is documented on page 13.*)

$\_\_hook\_use:wn$
$\_\_hook\_try\_file\_hook:n$
$\_\_hook\_if\_usable\_use:n$

$\_\_hook\_use:wn$ 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. $\_\_hook\_if\_usable\_use:n$ checks if the hook exist, and calls $\_\_hook\_preamble\_hook:n$ if so, then uses the hook.

```
1162 ⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_use:wn}
1163 ⟨latexrelease⟩                    {Standardise~generic~hook~names}
1164 ⟨latexrelease⟩\EndIncludeInRelease
1165 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_use:wn}
1166 ⟨latexrelease⟩                    {Standardise~generic~hook~names}
1167 ⟨latexrelease⟩\cs_new:Npn \__hook_use:wn #1 / #2 \s__hook_mark #3
1168 ⟨latexrelease⟩  {
1169 ⟨latexrelease⟩     \str_if_eq:nnTF {#1} { file }
1170 ⟨latexrelease⟩       { \__hook_try_file_hook:n {#3} }
1171 ⟨latexrelease⟩       { } % Hook doesn't exist
1172 ⟨latexrelease⟩  }

1173 ⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_file_hook:n #1
1174 ⟨latexrelease⟩  {
1175 ⟨latexrelease⟩     \__hook_if_file_hook:wTF #1 / \s__hook_mark
1176 ⟨latexrelease⟩       {
1177 ⟨latexrelease⟩          \exp_args:Ne \__hook_if_usable_use:n
1178 ⟨latexrelease⟩            { \exp_args:Ne \__hook_file_hook_normalize:n {#1} }
1179 ⟨latexrelease⟩       }
1180 ⟨latexrelease⟩       { \__hook_if_usable_use:n {#1} } % file/ generic hook (e.g. file/before)
1181 ⟨latexrelease⟩  }

1182 ⟨latexrelease⟩\cs_new_protected:Npn \__hook_if_usable_use:n #1
1183 ⟨latexrelease⟩  {
1184 ⟨latexrelease⟩     \tl_if_exist:cT { __hook~#1 }
1185 ⟨latexrelease⟩       {
1186 ⟨latexrelease⟩          \__hook_preamble_hook:n {#1}
1187 ⟨latexrelease⟩          \cs:w __hook~#1 \cs_end:
1188 ⟨latexrelease⟩       }
1189 ⟨latexrelease⟩  }
1190 ⟨latexrelease⟩\EndIncludeInRelease
```

(*End definition for* $\_\_hook\_use:wn$*,* $\_\_hook\_try\_file\_hook:n$*, and* $\_\_hook\_if\_usable\_use:n$*.*)

$\hook\_use\_once:n$
$\_\_hook\_use\_once:n$

For hooks that can and should be used only once we have a special use command that further inhibits the hook from getting more code added to it. This has the effect that any further code added to the hook is executed immediately rather than stored in the hook.

The code needs some gymnastics to prevent space trimming from the hook name, since $\hook\_use:n$ and $\hook\_use\_once:n$ are documented to not trim spaces.

```
1191 \cs_new_protected:Npn \hook_use_once:n #1
1192   {
1193     \__hook_if_execute_immediately:nF {#1}
1194       { \__hook_normalize_hook_args:Nn \__hook_use_once:n { \use:n {#1} } }
1195   }
1196 \cs_new_protected:Npn \__hook_use_once:n #1
```

```
1197    {
1198        \__hook_preamble_hook:n {#1}
1199        \__hook_use_once_set:n {#1}
1200        \__hook_use_initialized:n {#1}
1201        \__hook_use_once_clear:n {#1}
1202    }
```

__hook_use_once_set:n
__hook_use_once_clear:n

`\__hook_use_once_set:n` is used before the actual hook code is executed so that any usage of `\AddToHook` inside the hook causes the code to execute immediately. Setting `\g__hook_⟨hook⟩_reversed_tl` to `I` prevents further code from being added to the hook. `\__hook_use_once_clear:n` then clears the hook so that any further call to `\hook_use:n` or `\hook_use_once:n` will expand to nothing.

```
1203 \cs_new_protected:Npn \__hook_use_once_set:n #1
1204    { \__hook_tl_gset:cn { g__hook_#1_reversed_tl } { I } }
1205 \cs_new_protected:Npn \__hook_use_once_clear:n #1
1206    {
1207        \__hook_tl_gclear:c { __hook~#1 }
1208        \__hook_tl_gclear:c { __hook_next~#1 }
1209        \__hook_tl_gclear:c { __hook_toplevel~#1 }
1210        \prop_gclear:c { g__hook_#1_code_prop }
1211    }
```

(*End definition for* `\hook_use_once:n` *and others. This function is documented on page 13.*)

__hook_if_execute_immediately_p:n
__hook_if_execute_immediately:n*TF*

To check whether the code being added should be executed immediately (that is, if the hook is a one-time hook), we check if it's usable (it can't be one-time if it was not already usable), then we check that `\g__hook_⟨hook⟩_reversed_tl` is `I`. The gymnastics around `\if:w` is there to allow the `reversed` token list to be empty.

```
1212 \prg_new_conditional:Npnn \__hook_if_execute_immediately:n #1 { F, TF }
1213    {
1214        \__hook_if_usable:nTF {#1}
1215            {
1216                \exp_after:wN \__hook_clean_to_scan:w
1217                \if:w I \cs:w g__hook_#1_reversed_tl \cs_end:
1218                    \s__hook_mark \prg_return_true:
1219                \else:
1220                    \s__hook_mark \prg_return_false:
1221                \fi:
1222            }
1223            { \prg_return_false: }
1224    }
```

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

4.10 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 several other states as discussed in 4.4.2. 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) or may be disabled.

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).

All other states can only be queried with internal tests as the different states are irrelevant for package code.

\hook_if_empty_p:n
\hook_if_empty:n*TF*
Test if a hook is empty (that is, no code was added to that hook). A ⟨*hook*⟩ being empty means that all three of its \g__hook_⟨*hook*⟩_code_prop, its __hook_toplevel␣⟨*hook*⟩ and its __hook_next␣⟨*hook*⟩ are empty.

```
1225 \prg_new_conditional:Npnn \hook_if_empty:n #1 { p , T , F , TF }
1226   {
1227     \__hook_if_structure_exist:nTF {#1}
1228       {
1229         \bool_lazy_and:nnTF
1230           { \prop_if_empty_p:c { g__hook_#1_code_prop } }
1231           {
1232             \bool_lazy_and_p:nn
1233               { \tl_if_empty_p:c { __hook_toplevel~#1 } }
1234               { \tl_if_empty_p:c { __hook_next~#1 } }
1235           }
1236         { \prg_return_true: }
1237         { \prg_return_false: }
1238       }
1239       { \prg_return_true: }
1240   }
```

(*End definition for* \hook_if_empty:nTF. *This function is documented on page 14.*)

__hook_if_usable_p:n
__hook_if_usable:n*TF*
A hook is usable if the token list that stores the sorted code for that hook, __hook ⟨*hook*⟩, 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).

```
1241 \prg_new_conditional:Npnn \__hook_if_usable:n #1 { p , T , F , TF }
1242   {
1243     \tl_if_exist:cTF { __hook~#1 }
1244       { \prg_return_true: }
1245       { \prg_return_false: }
1246   }
```

(*End definition for* __hook_if_usable:nTF.)

__hook_if_structure_exist_p:n
__hook_if_structure_exist:n*TF*
An internal check if the hook has already its basic internal structure set up with __hook_init_structure: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.

```
1247 \prg_new_conditional:Npnn \__hook_if_structure_exist:n #1 { p , T , F , TF }
1248   {
1249     \prop_if_exist:cTF { g__hook_#1_code_prop }
1250       { \prg_return_true: }
1251       { \prg_return_false: }
1252   }
```

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

`\__hook_if_declared_p:n`
`\__hook_if_declared:nTF`   Internal test to check if the hook was officially declared with `\hook_new:n` or a variant.

```
1253 \prg_new_conditional:Npnn \__hook_if_declared:n #1 { p, T, F, TF }
1254   {
1255     \tl_if_exist:cTF { g__hook_#1_declared_tl }
1256       { \prg_return_true: }
1257       { \prg_return_false: }
1258   }
```

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

`\__hook_if_reversed_p:n`
`\__hook_if_reversed:nTF`   An internal conditional that checks if a hook is reversed.

```
1259 \prg_new_conditional:Npnn \__hook_if_reversed:n #1 { p , T , F , TF }
1260   {
1261     \exp_after:wN \__hook_clean_to_scan:w
1262     \if:w - \cs:w g__hook_#1_reversed_tl \cs_end:
1263       \s__hook_mark \prg_return_true:
1264     \else:
1265       \s__hook_mark \prg_return_false:
1266     \fi:
1267   }
```

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

`\__hook_if_generic_p:n`
`\__hook_if_generic:nTF`
`\__hook_if_deprecated_generic_p:n`
`\__hook_if_deprecated_generic:nTF`   An internal conditional that checks if a name belongs to a generic hook. The deprecated version needs to check if `#3` is empty to avoid returning true on `file/before`, for example.

```
1268 \prg_new_conditional:Npnn \__hook_if_generic:n #1 { T, TF }
1269   { \__hook_if_generic:w #1 / / / \s__hook_mark }
1270 \cs_new:Npn \__hook_if_generic:w #1 / #2 / #3 / #4 \s__hook_mark
1271   {
1272     \cs_if_exist:cTF { c__hook_generic_#1/./#3_tl }
1273       { \prg_return_true: }
1274       { \prg_return_false: }
1275   }
1276 \prg_new_conditional:Npnn \__hook_if_deprecated_generic:n #1 { T, TF }
1277   { \__hook_if_deprecated_generic:w #1 / / / \s__hook_mark }
1278 \cs_new:Npn \__hook_if_deprecated_generic:w #1 / #2 / #3 / #4 \s__hook_mark
1279   {
1280     \cs_if_exist:cTF { c__hook_deprecated_#1/./#2_tl }
1281       {
1282         \tl_if_empty:nTF {#3}
1283           { \prg_return_false: }
1284           { \prg_return_true: }
1285       }
1286       { \prg_return_false: }
1287   }
```

(*End definition for* `\__hook_if_generic:nTF` *and* `\__hook_if_deprecated_generic:nTF`.)

`\__hook_if_generic_reversed_p:n`
`\__hook_if_generic_reversed:nTF`   An internal conditional that checks if a name belongs to a generic reversed hook.

```
1288 \prg_new_conditional:Npnn \__hook_if_generic_reversed:n #1 { T }
1289   { \__hook_if_generic_reversed:w #1 / / / \scan_stop: }
1290 \cs_new:Npn \__hook_if_generic_reversed:w #1 / #2 / #3 / #4 \scan_stop:
```

```
1291    {
1292      \if_charcode:w - \cs:w c__hook_generic_#1/./#3_tl \cs_end:
1293        \prg_return_true:
1294      \else:
1295        \prg_return_false:
1296      \fi:
1297    }
```

(*End definition for* __hook_if_generic_reversed:nTF.)

4.11 Messages

Hook errors are LaTeX kernel errors:

```
1298 \prop_gput:Nnn \g_msg_module_type_prop { hooks } { LaTeX }
```

And so are kernel errors (this should move elsewhere eventually).

```
1299 \prop_gput:Nnn \g_msg_module_type_prop { latex2e } { LaTeX }
1300 \prop_gput:Nnn \g_msg_module_name_prop { latex2e } { kernel }
1301 \msg_new:nnnn { hooks } { labels-incompatible }
1302    {
1303      Labels~'#1'~and~'#2'~are~incompatible
1304      \str_if_eq:nnF {#3} {??} { ~in~hook~'#3' } .~
1305      \int_compare:nNnTF {#4} = { 1 }
1306        { The~ code~ for~ both~ labels~ will~ be~ dropped. }
1307        { You~ may~ see~ errors~ later. }
1308    }
1309    { LaTeX~found~two~incompatible~labels~in~the~same~hook.~
1310      This~indicates~an~incompatibility~between~packages.  }
1311 \msg_new:nnnn { hooks } { exists }
1312    { Hook~'#1'~ has~ already~ been~ declared. }
1313    { There~ already~ exists~ a~ hook~ declaration~ with~ this~
1314      name.\\
1315      Please~ use~ a~ different~ name~ for~ your~ hook.}
1316 \msg_new:nnnn { hooks } { hook-disabled }
1317    { Cannot~add~code~to~disabled~hook~'#1'. }
1318    {
1319      The~hook~'#1'~you~tried~to~add~code~to~was~previously~disabled~
1320      with~\iow_char:N\\hook_disable_generic:n~or~\iow_char:N\\DisableGenericHook,~so~
1321      it~cannot~have~code~added~to~it.
1322    }
1323 \msg_new:nnn { hooks } { empty-label }
1324    {
1325      Empty~code~label~\msg_line_context:.~
1326      Using~'\__hook_currname_or_default:'~instead.
1327    }
1328 \msg_new:nnn { hooks } { no-default-label }
1329    {
1330      Missing~(empty)~default~label~\msg_line_context:. \\
1331      This~command~was~ignored.
1332    }
```

```
1333 \msg_new:nnnn { hooks } { unknown-rule }
1334   { Unknown~ relationship~ '#3'~
1335     between~ labels~ '#2'~ and~ '#4'~
1336     \str_if_eq:nnF {#1} {??} { ~in~hook~'#1' }. ~
1337     Perhaps~ a~ misspelling?
1338   }
1339   {
1340     The~ relation~ used~ not~ known~ to~ the~ system.~ Allowed~ values~ are~
1341     'before'~ or~ '<',~
1342     'after'~ or~ '>',~
1343     'incompatible-warning',~
1344     'incompatible-error',~
1345     'voids'~ or~
1346     'unrelated'.
1347   }
1348 \msg_new:nnnn { hooks } { misused-top-level }
1349   {
1350     Illegal~use~of~\iow_char:N \\AddToHook{#1}[top-level]{...}.\\
1351     'top-level'~is~reserved~for~the~user's~document.
1352   }
1353   {
1354     The~'top-level'~label~is~meant~for~user~code~only,~and~should~only~
1355     be~used~(sparingly)~in~the~main~document.~Use~the~default~label~
1356     '\__hook_currname_or_default:'~for~this~\@cls@pkg,~or~another~
1357     suitable~label.
1358   }
1359 \msg_new:nnn { hooks } { set-top-level }
1360   {
1361     You~cannot~change~the~default~label~#1~'top-level'.~Illegal \\
1362     \use:nn { ~ } { ~ } \iow_char:N \\#2{#3} \\
1363     \msg_line_context:.
1364   }
1365 \msg_new:nnn { hooks } { extra-pop-label }
1366   {
1367     Extra~\iow_char:N \\PopDefaultHookLabel. \\
1368     This~command~will~be~ignored.
1369   }
1370 \msg_new:nnn { hooks } { missing-pop-label }
1371   {
1372     Missing~\iow_char:N \\PopDefaultHookLabel. \\
1373     The~label~'#1'~was~pushed~but~never~popped.~Something~is~wrong.
1374   }
1375 \msg_new:nnn { latex2e } { should-not-happen }
1376   {
1377     This~should~not~happen.~#1 \\
1378     Please~report~at~https://github.com/latex3/latex2e.
1379   }
1380 \msg_new:nnn { hooks } { activate-disabled }
1381   {
1382     Cannot~ activate~ hook~ '#1'~ because~ it~ is~ disabled!
1383   }
1384 \msg_new:nnn { hooks } { cannot-remove }
```

```
1385    {
1386      Cannot~remove~chunk~'#2'~from~hook~'#1'~because~
1387      \__hook_if_structure_exist:nTF {#1}
1388        { it~does~not~exist~in~that~hook. }
1389        { the~hook~does~not~exist. }
1390    }
1391 \msg_new:nnn { hooks } { generic-deprecated }
1392    {
1393      Generic~hook~'#1/#2/#3'~is~deprecated. \\
1394      Use~hook~'#1/#3/#2'~instead.
1395    }
```

4.12 LATEX 2ε package interface commands

\NewHook Declaring new hooks . . .

\NewReversedHook
\NewMirroredHookPair

```
1396 \NewDocumentCommand \NewHook            { m }{ \hook_new:n {#1} }
1397 \NewDocumentCommand \NewReversedHook    { m }{ \hook_new_reversed:n {#1} }
1398 \NewDocumentCommand \NewMirroredHookPair { mm }{ \hook_new_pair:nn {#1}{#2} }
```

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

```
1399 ⟨latexrelease⟩\IncludeInRelease{2021/06/01}%
1400 ⟨latexrelease⟩                {\hook_activate_generic:n}{Providing~hooks}
```

\ActivateGenericHook Providing new hooks . . .

```
1401 \NewDocumentCommand \ActivateGenericHook { m }{ \hook_activate_generic:n {#1} }
```

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

\DisableGenericHook Disabling a generic hook.

```
1402 \NewDocumentCommand \DisableGenericHook { m }{ \hook_disable_generic:n {#1} }
```

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

```
1403 ⟨latexrelease⟩\EndIncludeInRelease
1404 ⟨latexrelease⟩\IncludeInRelease{2020/10/01}
1405 ⟨latexrelease⟩                    {\hook_activate_generic:n}{Providing~hooks}
1406 ⟨latexrelease⟩
1407 ⟨latexrelease⟩\def \ActivateGenericHook#1{}
1408 ⟨latexrelease⟩
1409 ⟨latexrelease⟩\EndIncludeInRelease
```

\AddToHook

```
1410 \NewDocumentCommand \AddToHook { m o +m }
1411    { \hook_gput_code:nnn {#1} {#2} {#3} }
```

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

\AddToHookNext

```
1412 \NewDocumentCommand \AddToHookNext { m +m }
1413    { \hook_gput_next_code:nn {#1} {#2} }
```

(*End definition for* \AddToHookNext*. This function is documented on page 6.*)

\ClearHookNext

```
1414 \NewDocumentCommand \ClearHookNext { m }
1415   { \hook_gclear_next_code:n {#1} }
```

(*End definition for* \ClearHookNext*. This function is documented on page 6.*)

\RemoveFromHook

```
1416 \NewDocumentCommand \RemoveFromHook { m o }
1417   { \hook_gremove_code:nn {#1} {#2} }
```

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

\SetDefaultHookLabel
\PushDefaultHookLabel
\PopDefaultHookLabel

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

```
1418 \NewDocumentCommand \SetDefaultHookLabel { m }
1419   { \__hook_set_default_hook_label:n {#1} }
1420 %
1421 %   The label is only automatically updated with \cs{@onefilewithoptions}
1422 %   (\cs{usepackage} and \cs{documentclass}), but some packages, like
1423 %   Ti\emph{k}Z, define package-like interfaces, like
1424 %   \cs{usetikzlibrary} that are wrappers around \cs{input}, so they
1425 %   inherit the default label currently in force (usually |top-level|,
1426 %   but it may change if loaded in another package).  To provide a
1427 %   package-like behavior also for hooks in these files, we provide
1428 %   high-level access to the default label stack.
1429 %     \begin{macrocode}
1430 \NewDocumentCommand \PushDefaultHookLabel { m }
1431   { \__hook_curr_name_push:n {#1} }
1432 \NewDocumentCommand \PopDefaultHookLabel { }
1433   { \__hook_curr_name_pop: }
```

The current label stack holds the labels for all files but the current one (more or less like \@currnamestack), and the current label token list, \g__hook_hook_curr_name_tl, holds the label for the current file. However \@pushfilename happens before \@currname is set, so we need to look ahead to get the \@currname for the label. expl3 also requires the current file in \@pushfilename, so here we abuse \@expl@push@filename@aux@@ to do __hook_curr_name_push:n.

```
1434 \cs_gset_protected:Npn \@expl@push@filename@aux@@ #1#2#3
1435   {
1436     \__hook_curr_name_push:n {#3}
1437     \str_gset:Nx \g_file_curr_name_str {#3}
1438     #1 #2 {#3}
1439   }
```

(*End definition for* \SetDefaultHookLabel *,* \PushDefaultHookLabel *, and* \PopDefaultHookLabel*. These functions are documented on page 8.*)

\UseHook
\UseOneTimeHook

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

```
1440 \cs_new:Npn \UseHook        { \hook_use:n }
1441 \cs_new:Npn \UseOneTimeHook { \hook_use_once:n }
```

(*End definition for* \UseHook *and* \UseOneTimeHook*. These functions are documented on page 4.*)

\ShowHook
\LogHook

```
1442 \cs_new_protected:Npn \ShowHook { \hook_show:n }
1443 \cs_new_protected:Npn \LogHook { \hook_log:n }
```

(*End definition for* \ShowHook *and* \LogHook*. These functions are documented on page 11.*)

\DebugHooksOn
\DebugHooksOff

```
1444 \cs_new_protected:Npn \DebugHooksOn  { \hook_debug_on:  }
1445 \cs_new_protected:Npn \DebugHooksOff { \hook_debug_off: }
```

(*End definition for* \DebugHooksOn *and* \DebugHooksOff*. These functions are documented on page 12.*)

\DeclareHookRule

```
1446 \NewDocumentCommand \DeclareHookRule { m m m m }
1447                       { \hook_gset_rule:nnnn {#1}{#2}{#3}{#4} }
```

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

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

```
1448 \NewDocumentCommand \DeclareDefaultHookRule { m m m }
1449                       { \hook_gset_rule:nnnn {??}{#1}{#2}{#3} }
1450 \@onlypreamble\DeclareDefaultHookRule
```

(*End definition for* \DeclareDefaultHookRule*. This function is documented on page 10.*)

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

> *FMi: Needs perhaps an L3 interface, or maybe it should get dropped?*

```
1451 \NewDocumentCommand \ClearHookRule { m m m }
1452 { \hook_gset_rule:nnnn {#1}{#2}{unrelated}{#3} }
```

(*End definition for* \ClearHookRule*. This function is documented on page 10.*)

\IfHookEmptyTF Here we avoid the overhead of xparse, since \IfHookEmptyTF is used in \end (that is, every LATEX environment). As a further optimization, use \let rather than \def to avoid one expansion step.

```
1453 \cs_new_eq:NN \IfHookEmptyTF \hook_if_empty:nTF
```

(*End definition for* \IfHookEmptyTF*. This function is documented on page 10.*)

\IfHookExistsTF Marked for removal and no longer documented in the doc section!

> *PhO:* \IfHookExistsTF *is used in* jlreq.cls, pxatbegshi.sty, pxeverysel.sty, pxeveryshi.sty, *so the public name may be an alias of the internal conditional for a while. Regardless, those packages' use for* \IfHookExistsTF *is not really correct and can be changed.*

```
1454 \cs_new_eq:NN \IfHookExistsTF \__hook_if_usable:nTF
```

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

4.13 Deprecated that needs cleanup at some point

\hook_disable:n
\hook_provide:n
\hook_provide_reversed:n
\hook_provide_pair:nn
_hook_activate_generic_reversed:n
_hook_activate_generic_pair:nn

Deprecated.

```
1455 \cs_new_protected:Npn \hook_disable:n
1456   {
1457     \__hook_deprecated_warn:nn
1458       { hook_disable:n }
1459       { hook_disable_generic:n }
1460     \hook_disable_generic:n
1461   }
1462 \cs_new_protected:Npn \hook_provide:n
1463   {
1464     \__hook_deprecated_warn:nn
1465       { hook_provide:n }
1466       { hook_activate_generic:n }
1467     \hook_activate_generic:n
1468   }
1469 \cs_new_protected:Npn \hook_provide_reversed:n
1470   {
1471     \__hook_deprecated_warn:nn
1472       { hook_provide_reversed:n }
1473       { hook_activate_generic:n }
1474     \__hook_activate_generic_reversed:n
1475   }
1476 \cs_new_protected:Npn \hook_provide_pair:nn
1477   {
1478     \__hook_deprecated_warn:nn
1479       { hook_provide_pair:nn }
1480       { hook_activate_generic:n }
1481     \__hook_activate_generic_pair:nn
1482   }
1483 \cs_new_protected:Npn \__hook_activate_generic_reversed:n #1
1484   { \__hook_normalize_hook_args:Nn \__hook_activate_generic:nn {#1} { - } }
1485 \cs_new_protected:Npn \__hook_activate_generic_pair:nn #1#2
1486   { \hook_activate_generic:n {#1} \__hook_activate_generic_reversed:n {#2} }
```

(*End definition for* \hook_disable:n *and others. These functions are documented on page* **??**.)

\DisableHook
\ProvideHook
\ProvideReversedHook
\ProvideMirroredHookPair

Deprecated.

```
1487 \cs_new_protected:Npn \DisableHook
1488   {
1489     \__hook_deprecated_warn:nn
1490       { DisableHook }
1491       { DisableGenericHook }
1492     \hook_disable_generic:n
1493   }
1494 \cs_new_protected:Npn \ProvideHook
1495   {
1496     \__hook_deprecated_warn:nn
1497       { ProvideHook }
1498       { ActivateGenericHook }
1499     \hook_activate_generic:n
1500   }
1501 \cs_new_protected:Npn \ProvideReversedHook
```

```
1502    {
1503      \__hook_deprecated_warn:nn
1504        { ProvideReversedHook }
1505        { ActivateGenericHook }
1506      \__hook_activate_generic_reversed:n
1507    }
1508  \cs_new_protected:Npn \ProvideMirroredHookPair
1509    {
1510      \__hook_deprecated_warn:nn
1511        { ProvideMirroredHookPair }
1512        { ActivateGenericHook }
1513      \__hook_activate_generic_pair:nn
1514    }
```

(*End definition for* \DisableHook *and others. These functions are documented on page* **??**.)

__hook_deprecated_warn:nn   Warns about a deprecation, telling what should be used instead.

```
1515  \cs_new_protected:Npn \__hook_deprecated_warn:nn #1 #2
1516    { \msg_warning:nnnn { hooks } { deprecated } {#1} {#2} }
1517  \msg_new:nnn { hooks } { deprecated }
1518    {
1519      Command~\iow_char:N\\#1~is~deprecated~and~will~be~removed~in~a~
1520      future~release. \\ \\
1521      Use~\iow_char:N\\#2~instead.
1522    }
```

(*End definition for* __hook_deprecated_warn:nn.)

4.14   Internal commands needed elsewhere

Here we set up a few horrible (but consistent) LaTeX 2$_\varepsilon$ names to allow for internal commands to be used outside this module. We have to unset the @@ since we want double "at" sign in place of double underscores.

```
1523  ⟨@@=⟩
```

\@expl@@@initialize@all@@
\@expl@@@hook@curr@name@pop@@

```
1524  \cs_new_eq:NN \@expl@@@initialize@all@@
1525                \__hook_initialize_all:
```

```
1526  \cs_new_eq:NN \@expl@@@hook@curr@name@pop@@
1527                \__hook_curr_name_pop:
```

(*End definition for* \@expl@@@initialize@all@@ *and* \@expl@@@hook@curr@name@pop@@*. These functions are documented on page* **??**.)

Rolling back here doesn't undefine the interface commands as they may be used in packages without rollback functionality. So we just make them do nothing which may or may not work depending on the code usage.

```
1528  %
1529  ⟨latexrelease⟩\IncludeInRelease{0000/00/00}%
1530  ⟨latexrelease⟩                          {lthooks}{The~hook~management}%
1531  ⟨latexrelease⟩
1532  ⟨latexrelease⟩\def \NewHook#1{}
1533  ⟨latexrelease⟩\def \NewReversedHook#1{}
1534  ⟨latexrelease⟩\def \NewMirroredHookPair#1#2{}
```

1535 ⟨latexrelease⟩
1536 ⟨latexrelease⟩\def \DisableGenericHook #1{}
1537 ⟨latexrelease⟩
1538 ⟨latexrelease⟩\long\def\AddToHookNext#1#2{}
1539 ⟨latexrelease⟩
1540 ⟨latexrelease⟩\def\AddToHook#1{\@gobble@AddToHook@args}
1541 ⟨latexrelease⟩\providecommand\@gobble@AddToHook@args[2][]{}
1542 ⟨latexrelease⟩
1543 ⟨latexrelease⟩\def\RemoveFromHook#1{\@gobble@RemoveFromHook@arg}
1544 ⟨latexrelease⟩\providecommand\@gobble@RemoveFromHook@arg[1][]{}
1545 ⟨latexrelease⟩
1546 ⟨latexrelease⟩\def \UseHook   #1{}
1547 ⟨latexrelease⟩\def \UseOneTimeHook #1{}
1548 ⟨latexrelease⟩\def \ShowHook #1{}
1549 ⟨latexrelease⟩\let \DebugHooksOn \@empty
1550 ⟨latexrelease⟩\let \DebugHooksOff\@empty
1551 ⟨latexrelease⟩
1552 ⟨latexrelease⟩\def \DeclareHookRule #1#2#3#4{}
1553 ⟨latexrelease⟩\def \DeclareDefaultHookRule #1#2#3{}
1554 ⟨latexrelease⟩\def \ClearHookRule #1#2#3{}

If the hook management is not provided we make the test for existence false and the test for empty true in the hope that this is most of the time reasonable. If not a package would need to guard against running in an old kernel.

1555 ⟨latexrelease⟩\long\def \IfHookExistsTF #1#2#3{#3}
1556 ⟨latexrelease⟩\long\def \IfHookEmptyTF #1#2#3{#2}
1557 ⟨latexrelease⟩
1558 ⟨latexrelease⟩\EndModuleRelease
1559 \ExplSyntaxOff
1560 ⟨/2ekernel | latexrelease⟩

Index

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

78