LaTeX's hook management[*]

Frank Mittelbach[†]

2023 年 12 月 27 日

目录

1 简介 3

2 Package writer interface 3

2.1 LaTeX 2$_\varepsilon$ interfaces 3

2.1.1 Declaring hooks 3

2.1.2 Special declarations for generic hooks 4

2.1.3 Using hooks in code 5

2.1.4 Updating code for hooks 7

2.1.5 Hook names and default labels 10

2.1.6 The `top-level` label 13

2.1.7 Defining relations between hook code 13

2.1.8 Querying hooks 15

2.1.9 Displaying hook code 16

2.1.10 Debugging hook code 18

2.2 L3 programming layer (`expl3`) interfaces 18

2.3 On the order of hook code execution 22

2.4 The use of "reversed" hooks 24

2.5 Difference between "normal" and "one-time" hooks 25

2.6 Generic hooks provided by packages 26

2.7 Hooks with arguments 27

2.8 Private LaTeX kernel hooks 29

2.9 Legacy LaTeX 2$_\varepsilon$ interfaces 30

[*]This module has version v1.1f dated 2023/10/02, © LaTeX Project.

[†]Code improvements for speed and other goodies by Phelype Oleinik

3 LaTeX 2$_\varepsilon$ commands and environments augmented by hooks 32

3.1 Generic hooks . . . 32

3.1.1 Generic hooks for all environments . . . 33

3.1.2 Generic hooks for commands . . . 34

3.1.3 Generic hooks provided by file loading operations . . . 34

3.2 Hooks provided by `\begin{document}` . . . 35

3.3 Hooks provided by `\end{document}` . . . 36

3.4 Hooks provided by `\shipout` operations . . . 37

3.5 Hooks provided for paragraphs . . . 38

3.6 Hooks provided in NFSS commands . . . 38

3.7 Hook provided by the mark mechanism . . . 39

4 The Implementation 39

4.1 Debugging . . . 39

4.2 Borrowing from internals of other kernel modules . . . 40

4.3 Declarations . . . 40

4.4 Providing new hooks . . . 43

4.4.1 The data structures of a hook . . . 43

4.4.2 On the existence of hooks . . . 44

4.4.3 Setting hooks up . . . 46

4.4.4 Disabling and providing hooks . . . 53

4.5 Parsing a label . . . 55

4.6 Adding or removing hook code . . . 60

4.7 Setting rules for hooks code . . . 86

4.8 Specifying code for next invocation . . . 113

4.9 Using the hook . . . 116

4.10 Querying a hook . . . 123

4.11 Messages . . . 128

4.12 LaTeX 2$_\varepsilon$ package interface commands . . . 131

4.13 Deprecated that needs cleanup at some point . . . 136

4.14 Internal commands needed elsewhere . . . 138

1 介绍

钩子（Hooks）是命令或环境代码中的处理点，在这些点上可以添加处理代码到现有命令中。不同的包可以对同一命令进行处理，为了确保安全处理，需要将不同包添加的代码块按合适的顺序进行排序。

包通过 `\AddToHook` 添加代码块，并使用默认的包名作为标签对其进行标记。

在 `\begin{document}` 处，所有钩子的代码根据一些规则（由 `\DeclareHookRule` 给出）进行排序，以实现快速执行，避免额外的处理开销。如果后续修改了钩子代码（或更改了规则），将生成新的用于快速处理的版本。

一些钩子已在文档的导言部分使用。如果在此时已经使用了钩子，钩子将被准备（并排序）以便执行。

2 Package writer interface

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

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

`\NewHookWithArguments` `\NewHookWithArguments {⟨hook⟩} {⟨number⟩}`

Creates a new ⟨*hook*⟩ whose code takes ⟨*number*⟩ arguments, and otherwise works exactly like `\NewHook`. Section 2.7 explains hooks with arguments.

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

`\NewReversedHookWithArguments` `\NewReversedHookWithArguments {⟨hook⟩} {⟨number⟩}`

Like `\NewReversedHook`, but creates a hook whose code takes ⟨*number*⟩ arguments. Section 2.7 explains hooks with arguments.

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

`\NewMirroredHookPairWithArguments` `\NewMirroredHookPairWithArguments {⟨hook-1⟩} {⟨hook-2⟩} {⟨number⟩}`

A shorthand for `\NewHookWithArguments{`⟨*hook-1*⟩`}{`⟨*number*⟩`}`
`\NewReversedHookWithArguments{`⟨*hook-2*⟩`}{`⟨*number*⟩`}`. Section 2.7 explains hooks with arguments.

The ⟨*hook*⟩ 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 further attempt to add 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.

2.1.3 Using hooks in code

\UseHook `\UseHook {⟨hook⟩}`

Execute the code stored in the ⟨*hook*⟩.

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.

\UseHookWithArguments `\UseHookWithArguments {⟨hook⟩} {⟨number⟩} {⟨arg`$_1$`⟩} … {⟨arg`$_n$`⟩}`

Execute the code stored in the ⟨*hook*⟩ and pass the arguments `{`⟨*arg*$_1$⟩`}` through `{`⟨*arg*$_n$⟩`}` to the ⟨*hook*⟩. Otherwise, it works exactly like `\UseHook`. The ⟨*number*⟩ should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove ⟨*number*⟩ items from the input. Section 2.7 explains hooks with arguments.

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

[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.

`\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}`. From that point onwards, 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.

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`.

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

`\UseOneTimeHookWithArguments` `\UseOneTimeHookWithArguments {⟨hook⟩} {⟨number⟩} {⟨arg₁⟩} … {⟨argₙ⟩}`

Works exactly like `\UseOneTimeHook`, but passes arguments `{`⟨arg_1⟩`}` through `{`⟨arg_n⟩`}` to the ⟨*hook*⟩. The ⟨*number*⟩ should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove ⟨*number*⟩ items from the input.

It should be noted that after a one-time hook is used, it is no longer possible to use `\AddToHookWithArguments` or similar with that hook. `\AddToHook` continues to work as normal. Section 2.7 explains hooks with arguments.

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

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.

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

Works exactly like `\AddToHook`, except that the ⟨*code*⟩ can access the arguments passed to the hook using `#1`, `#2`, …, `#n` (up to the number of arguments declared for the hook). If the ⟨*code*⟩ should contain *parameter tokens* (`#`) that are not supposed to be understood as the arguments of the hook, such tokens should be doubled. For example, with `\AddToHook` one can write:

```
\AddToHook{myhook}{\def\foo#1{Hello, #1!}}
```

but to achieve the same with `\AddToHookWithArguments`, one should write:

```
\AddToHookWithArguments{myhook}{\def\foo##1{Hello, ##1!}}
```

because in the latter case, `#1` refers to the first argument of the hook `myhook`. Section 2.7 explains hooks with arguments.

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.

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

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

Works exactly like `\AddToHookNext`, but the ⟨*code*⟩ can contain references to the arguments of the ⟨*hook*⟩ as described for `\AddToHookWithArguments` above. Section 2.7 explains hooks with arguments.

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 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.

Important: *The dot-syntax is* ***not*** *available with* `\UseHook` *and some other commands that are typically used within code!*

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 the code in logical parts, but still use the main package name as the ⟨*label*⟩, then the ⟨*default label*⟩ can be set using `\PushDefaultHookLabel{...}`…`\PopDefaultHookLabel` or `\SetDefaultHookLabel{...}`.

`\PushDefaultHookLabel`
`\PopDefaultHookLabel`

```
\PushDefaultHookLabel {⟨default label⟩}
   ⟨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.

`\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

[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).

`\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.

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` `\ShowHook {`⟨*hook*⟩`}`
`\LogHook` `\LogHook {`⟨*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:

```
-> The hook 'example-hook':
> Code chunks:
>     foo -> [code from package 'foo']
>     bar -> [from package 'bar']
>     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 2_ε 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_new_with_args:nn`
`\hook_new_reversed_with_args:nn`
`\hook_new_pair_with_args:nnn`

```
\hook_new_with_args:nn {⟨hook⟩} {⟨number⟩}
\hook_new_reversed_with_args:nn {⟨hook⟩} {⟨number⟩}
\hook_new_pair_with_args:nnn {⟨hook-1⟩} {⟨hook-2⟩} {⟨number⟩}
```

Creates a new ⟨*hook*⟩ with normal or reverse ordering of code chunks, that takes ⟨*number*⟩ arguments from the input stream when it is used. `\hook_new_pair_-with_args: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⟩}`
`\hook_use:nnw` `\hook_use:nnw {⟨hook⟩} {⟨number⟩} {⟨arg`$_1$`⟩} … {⟨arg`$_n$`⟩}`

Executes the `{⟨hook⟩}` code followed (if set up) by the code for next invocation only, then empties that next invocation code. `\hook_use:nnw` should be used for hooks declared with arguments, and should be followed by as many brace groups as the declared number of arguments. The ⟨*number*⟩ should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove ⟨*number*⟩ items from the input.

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

`\hook_use_once:n` `\hook_use_once:n {⟨hook⟩}`
`\hook_use_once:nnw` `\hook_use_once:nnw {⟨hook⟩} {⟨number⟩} {⟨arg`$_1$`⟩} … {⟨arg`$_n$`⟩}`

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. `\hook_use_-once:nnw` should be used for hooks declared with arguments, and should be followed by as many brace groups as the declared number of arguments. The ⟨*number*⟩ should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove ⟨*number*⟩ items from the input.

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⟩}`
`\hook_gput_code_with_args:nnn` `\hook_gput_code_with_args: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 `\hook_gput_code_with_args:nnn` is used, the ⟨*code*⟩ can access the arguments passed to `\hook_use:nnw` (or `\hook_use_once:nnw`) with `#1`, `#2`, …, `#n` (up to the number of arguments declared for the hook). In that case, if an actual parameter token should be added to the code, it should be doubled.

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⟩}`
`\hook_gput_next_code_with_args:nn`

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

If `\hook_gput_next_code_with_args:nn` is used, the ⟨*code*⟩ can access the arguments passed to `\hook_use:nnw` (or `\hook_use_once:nnw`) with `#1`, `#2`, …, `#n` (up to the number of arguments declared for the hook). In that case, if an actual parameter token should be added to the code, it should be doubled.

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_if_empty:nTF {`⟨*hook*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

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

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

\hook_show:n
\hook_log:n

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

`\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_off:

`\hook_debug_on:`

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

2.3 On the order of hook code execution

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

Suppose you have the following declarations:

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

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

```
-> The hook 'myhook':
> Code chunks:
>     packageA -> \typeout {A}
>     packageB -> \typeout {B}
>     packageC -> \typeout {C}
> 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
```

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

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

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

and an error message saying that `\begin{color}` was 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):
>     ---
> 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.

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.

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

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 provides hooks such as `babel/`⟨*language*⟩`/afterextras`. However, 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 processing considerably.

2.7 Hooks with arguments

Sometimes it is necessary to pass contextual information to a hook, and, for one reason or another, it is not feasible to store such information in macros. To serve this purpose, hooks can be declared with arguments, so that the programmer can pass along the data necessary for the code in the hook to function properly.

A hook with arguments works mostly like a regular hook, and most commands that work for regular hooks, also work for hooks that take arguments. The differences are when the hook is declared (`\NewHookWithArguments` is used instead of `\NewHook`), then code can be added with both `\AddToHook` and `\AddToHookWithArguments`, and when the hook is used (`\UseHookWithArguments` instead of `\UseHook`).

A hook with arguments must be declared as such (before it is first used, as all regular hooks) using `\NewHookWithArguments{`⟨*hook*⟩`}{`⟨*number*⟩`}`. All code added to that hook can then use `#1` to access the first argument, `#2` to access the second, and so forth up to the number of arguments declared. However, it is still possible to add code with references to the arguments of a hook that was not yet declared (we will discuss that later). At their core, hooks are macros, so TEX's limit of 9 arguments applies, and a low-level TEX error is raised if you try to reference an argument number that doesn't exist.

To use a hook with arguments, just write `\UseHookWithArguments{`⟨*hook*⟩`}{`⟨*number*⟩`}` followed by a braced list of the arguments. For example, if the hook `test` takes three arguments, write:

```
\UseHookWithArguments{test}{3}{arg-1}{arg-2}{arg-3}
```

then, in the ⟨*code*⟩ of the hook, all instances of `#1` will be replaced by `arg-1`, `#2` by `arg-2` and so on. If, at the point of usage, the programmer provides more arguments than the hook is declared to take, the excess arguments are simply ignored by the hook. Behaviour is unpredictable[6] if too few arguments are provided. If the hook isn't declared, ⟨*number*⟩ arguments are removed from the input stream.

Adding code to a hook with arguments can be done with `\AddToHookWithArguments` as well as with the regular `\AddToHook`, to achieve different outcomes. The main difference when it comes to adding code to a hook, in this case, is firstly the possibility of accessing a hook's arguments, of course, and second, how parameter tokens (`#`$_6$) are treated.

Using `\AddToHook` in a hook that takes arguments will work as it does for all other hooks. This allows a package developer to add arguments to a hook that otherwise had none without having to worry about compatibility. This means that, for example:

```
\AddToHook{test}{\def\foo#1{Hello, #1!}}
```

will define the same macro `\foo` regardless if the hook `test` takes arguments or not.

Using `\AddToHookWithArguments` allows the ⟨*code*⟩ added to access the arguments of the hook with `#1`, `#2`, and so forth, up to the number of the arguments declared in the hook. This means that if one wants to add a `#`$_6$ to the ⟨*code*⟩ that token must be doubled in the input. The same definition from above, using `\AddToHookWithArguments`, needs to be rewritten:

[6]The hook *will* take the declared number of arguments, and what will happen depends on what was grabbed, and what the hook code does with its arguments.

```
\AddToHookWithArguments{test}{\def\foo##1{Hello, ##1!}}
```

Extending the above example to use the hook arguments, we could rewrite something like (now from declaration to usage, to get the whole picture):

```
\NewHookWithArguments{test}{1}
\AddToHookWithArguments{test}{%
  \typeout{Defining foo with "#1"}
  \def\foo##1{Hello, ##1! Some text after: #1}%
}
\UseHook{test}{Howdy!}
\ShowCommand\foo
```

Running the code above prints in the terminal:

```
Defining foo with "Howdy!"
> \foo=macro:
#1->Hello, #1! Some text after: Howdy!.
```

Note how `##1` in the call to `\AddToHookWithArguments` became `#1`, and the `#1` was replaced by the argument passed to the hook. Should the hook be used again, with a different argument, the definition would naturally change.

It is possible to add code referencing a hook's arguments before such hook is declared and the number of hooks is fixed. However, if some code is added to the hook, that references more arguments than will be declared for the hook, there will be a low-level TEX error about an "Illegal parameter number" at the time the hook is declared, which will be hard to track down because at that point TEX can't know whence the offending code came from. Thus it is important that package writers explicitly document how many arguments (if any) each hook can take, so users of those packages know how many arguments can be referenced, and equally important, what each argument means.

2.8 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 though 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.[7]

2.9 Legacy LaTeX 2$_\varepsilon$ interfaces

LaTeX 2$_\varepsilon$ 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.

[7]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!”

`\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 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.

The few hooks that existed previously in LaTeX 2$_\varepsilon$ 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.

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.[8] 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.

[8]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.

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.

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 by using `\AddToHook{begindocument}` or by using `\AtBeginDocument` and it is executed after the `.aux` file has been read 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$_\varepsilon$ has always provided `\AtEndDocument` to add code to the `\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 `\shipouts` 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).

3.7 Hook provided by the mark mechanism

See `ltmarks-doc.pdf` for details.

`insertmark` This hook allows for a special setup while `\InsertMark` inserts a mark. It is executed in group so local changes only apply to the mark being inserted.

4 The Implementation

```
⟨@@=hook⟩
⟨*2ekernel | latexrelease⟩
\ExplSyntaxOn
⟨latexrelease⟩\NewModuleRelease{2020/10/01}{lthooks}
⟨latexrelease⟩                    {The~hook~management~system}
```

4.1 Debugging

`\g__hook_debug_bool`

Holds the current debugging state.

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

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

`\hook_debug_on:`
`\hook_debug_off:`
`\__hook_debug:n`
`\__hook_debug_gset:`

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

```
\cs_new_eq:NN \__hook_debug:n \use_none:n
\cs_new_protected:Npn \hook_debug_on:
  {
    \bool_gset_true:N \g__hook_debug_bool
    \__hook_debug_gset:
  }
```

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

(*End of definition for* `\hook_debug_on:` *and others. These functions are documented on page 22.*)

4.2 Borrowing from internals of other kernel modules

`\__hook_str_compare:nn`

Private copy of `\__str_if_eq:nn`

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

(*End of definition for* `\__hook_str_compare:nn`*.*)

4.3 Declarations

`\l__hook_tmpa_bool`

Scratch boolean used throughout the package.

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

(*End of definition for* `\l__hook_tmpa_bool`*.*)

`\l__hook_return_tl`
`\l__hook_tmpa_tl`
`\l__hook_tmpb_tl`

Scratch variables used throughout the package.

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

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

`\g__hook_all_seq`

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

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

(*End of definition for* `\g__hook_all_seq`*.*)

`\l__hook_cur_hook_tl`

Stores the name of the hook currently being sorted.

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

(*End of 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.

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

(*End of definition for \l__hook_work_prop.*)

\g__hook_used_prop

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

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

(*End of definition for \g__hook_used_prop.*)

\g__hook_hook_curr_name_tl
\g__hook_name_stack_seq

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

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

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

__hook_tmp:w

Temporary macro for generic usage.

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

(*End of definition for __hook_tmp:w.*)

\c__hook_empty_tl
\c__hook_nine_parameters_tl

An empty token list, and one containing nine parameters.

```
\tl_const:Nn \c__hook_empty_tl { }
\tl_const:Nn \c__hook_nine_parameters_tl { #1#2#3#4#5#6#7#8#9 }
```

(*End of definition for \c__hook_empty_tl and \c__hook_nine_parameters_tl.*)

\tl_gremove_once:Nx
\tl_show:x
\tl_log:x
\tl_set:Ne
\cs_replacement_spec:c
\prop_put:Nne
\str_count:e

Some variants of expl3 functions.

FMi: should probably be moved to expl3

```
\cs_generate_variant:Nn \tl_gremove_once:Nn { Nx }
\cs_generate_variant:Nn \tl_show:n { x }
\cs_generate_variant:Nn \tl_log:n { x }
\cs_generate_variant:Nn \tl_set:Nn { Ne }
\cs_generate_variant:Nn \cs_replacement_spec:N { c }
\cs_generate_variant:Nn \prop_put:Nnn { Nne }
\cs_generate_variant:Nn \str_count:n { e }
```

(*End of definition for \tl_gremove_once:Nx and others.*)

\s__hook_mark

Scan mark used for delimited arguments.

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

(*End of definition for \s__hook_mark.*)

__hook_use_none_delimit_by_s_mark:w
__hook_use_i_delimit_by_s_mark:nw

Removes tokens until the next `\s__hook_mark`.

```
\cs_new:Npn \__hook_use_none_delimit_by_s_mark:w #1 \s__hook_mark { }
\cs_new:Npn \__hook_use_i_delimit_by_s_mark:nw #1 #2 \s__hook_mark {#1}
```

(*End of definition for* `\__hook_use_none_delimit_by_s_mark:w` *and* `\__hook_use_i_delimit_by_s_mark:nw`.)

__hook_tl_set:cn

Private copies of a few expl3 functions. l3debug will only add debugging to the public names, not to these copies, so we don't have to use `\debug_suspend:` and `\debug_resume:` everywhere.

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.

```
\cs_new_protected:Npn \__hook_tl_set:cn #1#2
  { \cs_set_nopar:cpx {#1} { \__kernel_exp_not:w {#2} } }
```

(*End of definition for* `\__hook_tl_set:cn`.)

__hook_tl_gset:Nn
__hook_tl_gset:Nx
__hook_tl_gset:cn
__hook_tl_gset:co
__hook_tl_gset:cx

Same as above.

```
\cs_new_protected:Npn \__hook_tl_gset:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \__kernel_exp_not:w {#2} } }
\cs_new_protected:Npn \__hook_tl_gset:Nx #1#2
  { \cs_gset_nopar:Npx #1 {#2} }
\cs_generate_variant:Nn \__hook_tl_gset:Nn { c, co }
\cs_generate_variant:Nn \__hook_tl_gset:Nx { c }
```

(*End of definition for* `\__hook_tl_gset:Nn`.)

__hook_tl_gput_right:Nn
__hook_tl_gput_right:Ne
__hook_tl_gput_right:cn

Same as above.

```
\cs_new_protected:Npn \__hook_tl_gput_right:Nn #1#2
  { \__hook_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
\cs_generate_variant:Nn \__hook_tl_gput_right:Nn { Ne, cn }
```

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

__hook_tl_gput_left:Nn

Same as above.

```
\cs_new_protected:Npn \__hook_tl_gput_left:Nn #1#2
  {
    \__hook_tl_gset:Nx #1
      { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
  }
```

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

__hook_tl_gset_eq:NN

Same as above.

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

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

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

Same as above.

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

(*End of 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⟩`
`\g_@@_⟨hook⟩_reversed_tl`
`\g_@@_⟨hook⟩_declared_tl`
`\g_@@_⟨hook⟩_parameter_tl`
`\@@_next␣⟨hook⟩`
`\@@_toplevel␣⟨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.

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

`\c__hook_⟨hook⟩_parameter_tl` This token list stores the parameter text for a declared hook (its existence almost completely intersects the token list above), which is used for managing hooks with arguments.

`\__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_with_args:nn
__hook_new:nn

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

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_new_with_args:nn}
⟨latexrelease⟩                    {Hooks~with~args}
\cs_new_protected:Npn \hook_new:n #1
  { \__hook_normalize_hook_args:Nn \__hook_new:nn {#1} { 0 } }
\cs_new_protected:Npn \hook_new_with_args:nn #1 #2
  { \__hook_normalize_hook_args:Nn \__hook_new:nn {#1} {#2} }
\cs_new_protected:Npn \__hook_new:nn #1 #2
  {
```

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.

```
    \__hook_if_declared:nTF {#1}
      { \msg_error:nnn { hooks } { exists } {#1} }
      {
        \tl_new:c { g__hook_#1_declared_tl }
        \cs_undefine:c { __hook~#1 }
        \cs_undefine:c { c__hook_#1_parameter_tl }
        \__hook_make_usable:nn {#1} {#2}
```

In case there is already code in a hook, but it's undeclared, run __hook_update_-hook_code:n to make it ready to be executed (see test lthooks-034).

```
        \__hook_update_hook_code:n {#1}
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_new_with_args:nn}
⟨latexrelease⟩                    {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \hook_new:n #1
⟨latexrelease⟩  { \__hook_normalize_hook_args:Nn \__hook_new:n {#1} }
⟨latexrelease⟩\cs_undefine:N \__hook_new:nn
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_new:n #1
⟨latexrelease⟩  {
```

```
⟨latexrelease⟩      \__hook_if_declared:nTF {#1}
⟨latexrelease⟩        { \msg_error:nnn { hooks } { exists } {#1} }
⟨latexrelease⟩        {
⟨latexrelease⟩          \tl_new:c { g__hook_#1_declared_tl }
⟨latexrelease⟩          \__hook_make_usable:n {#1}
⟨latexrelease⟩        }
⟨latexrelease⟩    }
⟨latexrelease⟩\cs_gset_protected:Npn \hook_new_with_args:nn #1 { }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_new:n`*,* `\hook_new_with_args:nn`*, and* `__hook_new:nn`*. These functions are documented on page 18.*)

`\__hook_make_usable:nn` 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:nnn` when adding code to a generic hook.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_make_usable:nn}
⟨latexrelease⟩                    {Hooks~with~args}
\cs_new_protected:Npn \__hook_make_usable:nn #1 #2
  {
```

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.

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

Here we'll define the `\c__hook_`⟨*hook*⟩`_parameter_tl` to hold a run of parameters up to the number of arguments of the hook (`#2`).

```
        \__kernel_cs_parm_from_arg_count:nnF
          { \tl_const:cn { c__hook_#1_parameter_tl } } {#2}
          {
            \msg_error:nnnn { hooks } { too-many-args } {#1} {#2}
            \tl_const:cx { c__hook_#1_parameter_tl }
              { \exp_not:V \c__hook_nine_parameters_tl }
          }
```

After that, use `\__hook_normalise_cs_args:nn` to correct the number of parameters of the macros `\__hook_toplevel␣`⟨*hook*⟩ and `\__hook_next␣`⟨*hook*⟩. We need to be able to add code with arguments to a hook without prior knowledge of the number of arguments of that hook, so lthooks assumes 9 until the hook is properly declared and the number of arguments is known. `\__hook_normalise_cs_args:nn` does the normalisation by using the `\c__hook_`⟨*hook*⟩`_parameter_tl` defined just above.

```
        \__hook_normalise_cs_args:nn { _toplevel } {#1}
        \__hook_normalise_cs_args:nn { _next } {#1}
```

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

```
        \__hook_code_gset:nn {#1} { }
```

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

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

The call to `\__hook_normalise_code_pool:n` will correct any improper reference to arguments that don't exist in the hook, raising a low-level TEX error and doubling the offending parameter tokens. It has to be done after `\__hook_init_structure:n` because it operates on `\g__hook_`⟨*hook*⟩`_code_prop`.

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

The `\g__hook_`⟨*hook*⟩`_labels_clist` holds the sorted list of labels (once it got sorted). This is used only for debugging. These are defined conditionally, in case `\__hook_make_usable:nn` is being used to redefine a hook.

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

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

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

The above is all in L3 convention, but we also provide an interface to legacy LaTeX 2_ε 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!

```
        \__hook_include_legacy_code_chunk:n {#1}
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_make_usable:nn}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_make_usable:nn
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_make_usable:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \tl_if_exist:cF { __hook~#1 }
⟨latexrelease⟩      {
⟨latexrelease⟩        \seq_gput_right:Nn \g__hook_all_seq {#1}
⟨latexrelease⟩        \tl_new:c { __hook~#1 }
⟨latexrelease⟩        \__hook_init_structure:n {#1}
⟨latexrelease⟩        \clist_new:c { g__hook_#1_labels_clist }
⟨latexrelease⟩        \tl_new:c { g__hook_#1_reversed_tl }
⟨latexrelease⟩        \__hook_include_legacy_code_chunk:n {#1}
⟨latexrelease⟩      }
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_make_usable:nn`.)

`\__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`.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_init_structure:n}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new_protected:Npn \__hook_init_structure:n #1
  {
    \__hook_if_structure_exist:nF {#1}
      {
        \prop_new:c { g__hook_#1_code_prop }
        \__hook_toplevel_gset:nn {#1} { }
        \__hook_next_gset:nn {#1} { }
      }
  }
⟨latexrelease⟩\EndIncludeInRelease

⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_init_structure:n}
⟨latexrelease⟩                  {Hooks~with~args}
```

```
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_init_structure:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_structure_exist:nF {#1}
⟨latexrelease⟩      {
⟨latexrelease⟩        \prop_new:c { g__hook_#1_code_prop }
⟨latexrelease⟩        \tl_new:c { __hook_toplevel~#1 }
⟨latexrelease⟩        \tl_new:c { __hook_next~#1 }
⟨latexrelease⟩      }
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
```

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

\hook_new_reversed:n
\hook_new_reversed_with_args:nn
__hook_new_reversed:nn

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

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_new_reversed_with_args:nn}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new_protected:Npn \hook_new_reversed:n #1
  { \__hook_normalize_hook_args:Nn \__hook_new_reversed:nn {#1} { 0 } }
\cs_new_protected:Npn \hook_new_reversed_with_args:nn #1 #2
  { \__hook_normalize_hook_args:Nn \__hook_new_reversed:nn {#1} {#2} }
\cs_new_protected:Npn \__hook_new_reversed:nn #1 #2
  {
    \__hook_if_declared:nTF {#1}
      { \msg_error:nnn { hooks } { exists } {#1} }
      {
        \__hook_new:nn {#1} {#2}
        \tl_gset:cn { g__hook_#1_reversed_tl } { - }
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_new_reversed_with_args:nn}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \hook_new_reversed:n #1
⟨latexrelease⟩  { \__hook_normalize_hook_args:Nn \__hook_new_reversed:n {#1} }
⟨latexrelease⟩\cs_undefine:N \__hook_new_reversed:nn
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_new_reversed:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_new:n {#1}
⟨latexrelease⟩    \tl_gset:cn { g__hook_#1_reversed_tl } { - }
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_undefine:N \__hook_new_reversed:nn
```

```
⟨latexrelease⟩\cs_gset_protected:Npn \hook_new_reversed_with_args:nn #1 #2 { }
⟨latexrelease⟩\EndIncludeInRelease
```

(End of definition for `\hook_new_reversed:n`, `\hook_new_reversed_with_args:nn`, and `\__hook_new_reversed:nn`. These functions are documented on page 18.)

`\hook_new_pair:nn`
`\hook_new_pair_with_args:nnn`

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

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_new_pair_with_args:nnn}
⟨latexrelease⟩                              {Hooks~with~args}
\cs_new_protected:Npn \hook_new_pair:nn #1#2
  { \__hook_normalize_hook_args:Nnn \__hook_new_pair:nnn {#1} {#2} { 0 } }
\cs_new_protected:Npn \hook_new_pair_with_args:nnn #1#2#3
  { \__hook_normalize_hook_args:Nnn \__hook_new_pair:nnn {#1} {#2} {#3} }
\cs_new_protected:Npn \__hook_new_pair:nnn #1 #2 #3
  {
    \__hook_if_declared:nTF {#1}
      { \msg_error:nnn { hooks } { exists } {#1} }
      {
        \__hook_if_declared:nTF {#2}
          { \msg_error:nnn { hooks } { exists } {#2} }
          {
            \__hook_new:nn {#1} {#3}
            \__hook_new_reversed:nn {#2} {#3}
          }
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_new_pair_with_args:nnn}
⟨latexrelease⟩                              {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \hook_new_pair:nn #1#2
⟨latexrelease⟩  {
⟨latexrelease⟩    \hook_new:n {#1}
⟨latexrelease⟩    \hook_new_reversed:n {#2}
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_gset_protected:Npn \hook_new_pair_with_args:nnn #1#2#3
⟨latexrelease⟩  { }
⟨latexrelease⟩\EndIncludeInRelease
```

(End of definition for `\hook_new_pair:nn` and `\hook_new_pair_with_args:nnn`. These functions are documented on page 18.)

`\__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!

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_include_legacy_code_chunk:n}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new_protected:Npn \__hook_include_legacy_code_chunk:n #1
  {
```

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

```
    \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.

```
        \tl_if_empty:cF { @#1hook }
          {
```

Here we set `\__hook_replacing_args_false:` because no legacy code will reference hook arguments.

```
            \__hook_replacing_args_false:
            \use:e
              {
                \__hook_hook_gput_code_do:nnn {#1} { legacy }
                  { \exp_not:v { @#1hook } }
              }
            \__hook_replacing_args_reset:
```

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

```
            \__hook_tl_gclear:c { @#1hook }
          }
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_include_legacy_code_chunk:n}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_include_legacy_code_chunk:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \tl_if_exist:cT { @#1hook }
⟨latexrelease⟩      {
⟨latexrelease⟩        \tl_if_empty:cF { @#1hook }
```

```
⟨latexrelease⟩            {
⟨latexrelease⟩              \exp_args:Nnnv \__hook_hook_gput_code_do:nnn
⟨latexrelease⟩                {#1} { legacy } { @#1hook }
⟨latexrelease⟩              \__hook_tl_gclear:c { @#1hook }
⟨latexrelease⟩            }
⟨latexrelease⟩        }
⟨latexrelease⟩    }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of 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.

```
⟨latexrelease⟩\IncludeInRelease{2021/06/01}{\hook_disable_generic:n}
⟨latexrelease⟩                 {Disable~hooks}
\cs_new_protected:Npn \hook_disable_generic:n #1
  { \__hook_normalize_hook_args:Nn \__hook_disable:n {#1} }
\cs_new_protected:Npn \__hook_disable:n #1
  {
    \tl_gclear_new:c { g__hook_#1_declared_tl }
    \cs_undefine:c { __hook~#1 }
  }
\prg_new_conditional:Npnn \__hook_if_disabled:n #1 { p, T, F, TF }
  {
    \bool_lazy_and:nnTF
        { \tl_if_exist_p:c { g__hook_#1_declared_tl } }
        { ! \cs_if_exist_p:c { __hook~#1 } }
      { \prg_return_true: }
      { \prg_return_false: }
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_disable_generic:n}
⟨latexrelease⟩                 {Disable~hooks}
⟨latexrelease⟩
⟨latexrelease⟩\cs_new_protected:Npn \hook_disable_generic:n #1 {}
```

```
⟨latexrelease⟩
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_disable_generic:n`*,* `\__hook_disable:n`*, and* `\__hook_if_disabled:nTF`*. This function is documented on page 19.*)

`\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.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_activate_generic:n}
⟨latexrelease⟩                 {Providing~hooks}
\cs_new_protected:Npn \hook_activate_generic:n #1
  { \__hook_normalize_hook_args:Nn \__hook_activate_generic:nn {#1} {   } }
\cs_new_protected:Npn \__hook_activate_generic:nn #1 #2
  {
```

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

```
    \__hook_if_disabled:nTF {#1}
      { \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.

```
      {
        \__hook_if_declared:nF {#1}
          {
            \tl_new:c { g__hook_#1_declared_tl }
            \__hook_make_usable:nn {#1} { 0 }
            \tl_gset:cx { g__hook_#1_reversed_tl }
              { \__hook_if_generic_reversed:nT {#1} { - } }
```

Reflect that we have activated the generic hook and set its execution code.

```
            \__hook_update_hook_code:n {#1}
          }
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2021/06/01}{\hook_activate_generic:n}
⟨latexrelease⟩                 {Providing~hooks}
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_activate_generic:nn #1 #2
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_disabled:nTF {#1}
⟨latexrelease⟩      { \msg_warning:nnn { hooks } { activate-disabled } {#1} }
```

```
⟨latexrelease⟩        {
⟨latexrelease⟩          \__hook_if_declared:nF {#1}
⟨latexrelease⟩            {
⟨latexrelease⟩              \tl_new:c { g__hook_#1_declared_tl }
⟨latexrelease⟩              \__hook_make_usable:n {#1}
⟨latexrelease⟩              \tl_gset:cx { g__hook_#1_reversed_tl }
⟨latexrelease⟩                { \__hook_if_generic_reversed:nT {#1} { - } }
⟨latexrelease⟩              \__hook_update_hook_code:n {#1}
⟨latexrelease⟩            }
⟨latexrelease⟩        }
⟨latexrelease⟩    }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_activate_generic:n}
⟨latexrelease⟩                      {Providing~hooks}
⟨latexrelease⟩\cs_gset_protected:Npn \hook_activate_generic:n #1 { }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_activate_generic:n` *and* `\__hook_activate_generic:n`. *This function is documented on page 19.*)

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:`.

```
\cs_new:Npn \__hook_parse_label_default:n #1
  {
    \tl_if_novalue:nTF {#1}
      { \__hook_currname_or_default: }
      { \tl_trim_spaces_apply:nN {#1} \__hook_parse_dot_label:n }
  }
```

(*End of 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.

```
\cs_new:Npn \__hook_parse_dot_label:n #1
  {
    \tl_if_empty:nTF {#1}
      {
        \msg_expandable_error:nn { hooks } { empty-label }
        \__hook_currname_or_default:
      }
      {
        \str_if_eq:nnTF {#1} { . }
          { \__hook_currname_or_default: }
          { \__hook_parse_dot_label:w #1 ./ \s__hook_mark }
      }
  }
\cs_new:Npn \__hook_parse_dot_label:w #1 ./ #2 \s__hook_mark
  {
    \tl_if_empty:nTF {#1}
      { \__hook_parse_dot_label_aux:w #2 \s__hook_mark }
      {
        \tl_if_empty:nTF {#2}
          { \__hook_make_name:n {#1} }
          { \__hook_parse_dot_label_cleanup:w #1 ./ #2 \s__hook_mark }
      }
  }
\cs_new:Npn \__hook_parse_dot_label_cleanup:w #1 ./ \s__hook_mark {#1}
\cs_new:Npn \__hook_parse_dot_label_aux:w #1 ./ \s__hook_mark
  { \__hook_currname_or_default: / \__hook_make_name:n {#1} }
```

(*End of 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`.

```
\cs_new:Npn \__hook_currname_or_default:
  {
    \tl_if_empty:NTF \g__hook_hook_curr_name_tl
      {
        \tl_if_empty:NTF \@currname
          {
            \msg_expandable_error:nnn { latex2e } { should-not-happen }
              { Empty~default~label. }
            \__hook_make_name:n { label-missing }
          }
          { \@currname }
      }
      { \g__hook_hook_curr_name_tl }
  }
```

(*End of 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.

```
\cs_new:Npn \__hook_make_name:n #1
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \__hook_make_name:w
    \exp_after:wN \token_to_str:N \cs:w __hook~ #1 \cs_end:
  }
\exp_last_unbraced:NNNNo
\cs_new:Npn \__hook_make_name:w #1 \tl_to_str:n { __hook~ } { }
```

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

__hook_normalize_hook_args:Nn
__hook_normalize_hook_args:Nnn
__hook_normalize_hook_rule_args:Nnnnn
__hook_normalize_hook_args_aux:Nn

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`!

```
\cs_new_protected:Npn \__hook_normalize_hook_args_aux:Nn #1 #2
  {
    \group_begin:
    \use:e
      {
        \group_end:
        \exp_not:N #1 #2
      }
  }
\cs_new_protected:Npn \__hook_normalize_hook_args:Nn #1 #2
  {
    \__hook_normalize_hook_args_aux:Nn #1
      { { \__hook_parse_label_default:n {#2} } }
  }
\cs_new_protected:Npn \__hook_normalize_hook_args:Nnn #1 #2 #3
  {
    \__hook_normalize_hook_args_aux:Nn #1
      {
        { \__hook_parse_label_default:n {#2} }
        { \__hook_parse_label_default:n {#3} }
      }
  }
\cs_new_protected:Npn \__hook_normalize_hook_rule_args:Nnnnn #1 #2 #3 #4 #5
  {
    \__hook_normalize_hook_args_aux:Nn #1
      {
        { \__hook_parse_label_default:n {#2} }
        { \__hook_parse_label_default:n {#3} }
        { \tl_trim_spaces:n {#4} }
        { \__hook_parse_label_default:n {#5} }
      }
  }
```

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

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

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`:

```
\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:

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

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

```
⟨latexrelease⟩\cs_set_protected:Npn \__hook_tmp:w #1 #2 #3
⟨latexrelease⟩  {
⟨latexrelease⟩    \quark_if_recursion_tail_stop:n {#1}
⟨latexrelease⟩    \seq_gput_right:Nn \g__hook_name_stack_seq {#1}
⟨latexrelease⟩    \__hook_tmp:w
⟨latexrelease⟩  }
⟨latexrelease⟩\exp_after:wN \__hook_tmp:w \@currnamestack
⟨latexrelease⟩  \q_recursion_tail \q_recursion_tail
⟨latexrelease⟩  \q_recursion_tail \q_recursion_stop
```

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

```
⟨latexrelease⟩\tl_gset:Nx \g__hook_hook_curr_name_tl { \@currname }
⟨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):

```
\cs_new_protected:Npn \__hook_curr_name_push:n #1
  { \exp_args:Nx \__hook_curr_name_push_aux:n { \__hook_make_name:n {#1} } }
\cs_new_protected:Npn \__hook_curr_name_push_aux:n #1
  {
    \tl_if_blank:nTF {#1}
      { \msg_error:nn { hooks } { no-default-label } }
      {
        \str_if_eq:nnTF {#1} { top-level }
          {
            \msg_error:nnnnn { hooks } { set-top-level }
              { to } { PushDefaultHookLabel } {#1}
          }
          {
            \seq_gpush:NV \g__hook_name_stack_seq \g__hook_hook_curr_name_tl
            \tl_gset:Nn \g__hook_hook_curr_name_tl {#1}
          }
      }
  }
```

and when an input is over, the topmost item of the stack is popped, since 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:

```
\cs_new_protected:Npn \__hook_curr_name_pop:
  {
    \seq_gpop:NNTF \g__hook_name_stack_seq \l__hook_return_tl
      { \tl_gset_eq:NN \g__hook_hook_curr_name_tl \l__hook_return_tl }
      { \msg_error:nn { hooks } { extra-pop-label } }
  }
```

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):

```
\tl_gput_right:Nn \@kernel@after@enddocument@afterlastpage
  { \__hook_end_document_label_check: }
\cs_new_protected:Npn \__hook_end_document_label_check:
  {
    \seq_gpop:NNT \g__hook_name_stack_seq \l__hook_return_tl
      {
        \msg_error:nnx { hooks } { missing-pop-label }
          { \g__hook_hook_curr_name_tl }
        \tl_gset_eq:NN \g__hook_hook_curr_name_tl \l__hook_return_tl
        \__hook_end_document_label_check:
      }
  }
```

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

__hook_set_default_hook_label:n
__hook_set_default_label:n

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

```
\cs_new_protected:Npn \__hook_set_default_hook_label:n #1
  {
    \seq_if_empty:NTF \g__hook_name_stack_seq
      {
        \msg_error:nnnnn { hooks } { set-top-level }
          { for } { SetDefaultHookLabel } {#1}
      }
      { \exp_args:Nx \__hook_set_default_label:n { \__hook_make_name:n {#1} } }
  }
\cs_new_protected:Npn \__hook_set_default_label:n #1
  {
    \str_if_eq:nnTF {#1} { top-level }
      {
        \msg_error:nnnnn { hooks } { set-top-level }
          { to } { SetDefaultHookLabel } {#1}
      }
      { \tl_gset:Nn \g__hook_hook_curr_name_tl {#1} }
  }
```

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

4.6 Adding or removing hook code

\hook_gput_code:nnn
\hook_gput_code_with_args:nnn
__hook_gput_code:nnn
__hook_gput_code_store:nnn
__hook_hook_gput_code_do:nnn
__hook_prop_gput_labeled_cleanup:nnn
__hook_prop_gput_labeled_do:Nnnn

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

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_gput_code:nnn}
⟨latexrelease⟩                        {Hooks~with~args}
\cs_new_protected:Npn \hook_gput_code:nnn #1 #2 #3
```

```
  {
    \__hook_replacing_args_false:
    \__hook_normalize_hook_args:Nnn \__hook_gput_code:nnn {#1} {#2} {#3}
    \__hook_replacing_args_reset:
  }
\cs_new_protected:Npn \hook_gput_code_with_args:nnn #1 #2 #3
  {
    \__hook_replacing_args_true:
    \__hook_normalize_hook_args:Nnn \__hook_gput_code:nnn {#1} {#2} {#3}
    \__hook_replacing_args_reset:
  }
```

If \AddToHookWithArguments was used, do some sanity checking, and if it's not possible to use arguments at this point, fall back to regular \AddToHook by using __hook_replacing_args_false:.

```
\cs_new_protected:Npn \__hook_gput_code:nnn #1 #2 #3
  {
    \__hook_chk_args_allowed:nn {#1} { AddToHook }
```

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

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

\AddToHookWithArguments can't be used on one-time hooks (that were already used).

```
        \__hook_if_replacing_args:TF
          {
            \msg_error:nnnn { hooks } { one-time-args }
              {#1} { AddToHook }
          }
          { }
        \use:n
      }
      { \__hook_gput_code_store:nnn {#1} {#2} }
          {#3}
  }
\cs_new_protected:Npn \__hook_gput_code_store:nnn #1 #2 #3
  {
```

Then check if the hook is usable.

```
    \__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.

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

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

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

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.

```
        {
          \__hook_if_disabled:nTF {#1}
            { \msg_error:nnn { hooks } { hook-disabled } {#1} }
            { \__hook_try_declaring_generic_hook:nnn {#1} {#2} {#3} }
        }
    }
```

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

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

However, first some debugging info if debugging is enabled:

```
      \__hook_debug:n{\iow_term:x{****~ Add~ to~
                          \__hook_if_usable:nF {#1} { undeclared~ }
                          hook~ #1~ (#2)
                          \on@line\space <-~ \tl_to_str:n{#3}} }
```

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

```
      \str_if_eq:nnTF {#2} { top-level }
        {
          \str_if_eq:eeTF { top-level } { \__hook_currname_or_default: }
            {
```

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

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

Then append to the `_toplevel` container for the hook.

```
            \__hook_cs_gput_right:nnn { _toplevel } {#1} {#3}
          }
          { \msg_error:nnn { hooks } { misused-top-level } {#1} }
      }
      {
```

When adding to the code pool, we have to double hashes if `\AddToHook` was used (`replacing_args` is false), so that later it is turned into a single parameter token, rather than a parameter to the hook macro.

```
        \exp_args:Nx \__hook_prop_gput_labeled_cleanup:nnn
          {
            \__hook_if_replacing_args:TF
              { \exp_not:n }
              { \__hook_double_hashes:n }
                {#3}
          }
          {#1} {#2}
      }
  }
```

Adds code to a hook's code pool.

```
\cs_new_protected:Npn \__hook_prop_gput_labeled_cleanup:nnn #1 #2 #3
  {
    \tl_set:Nn \l__hook_return_tl {#1}
    \__hook_if_replacing_args:TF
      {
        \__hook_if_usable:nT {#2}
          {
            \__hook_set_normalise_fn:nn {#2}
              { Invalid~code~added~\msg_line_context: }
            \__hook_normalise_fn:nn {#3} {#1}
            \prop_get:NnN \l__hook_work_prop {#3} \l__hook_return_tl
          }
      }
      { }
    \exp_args:NcV \__hook_prop_gput_labeled_do:Nnn
      { g__hook_#2_code_prop } \l__hook_return_tl {#3}
  }
\cs_new_protected:Npn \__hook_prop_gput_labeled_do:Nnn #1 #2 #3
  {
    \prop_get:NnNTF #1 {#3} \l__hook_return_tl
      { \prop_gput:Nno #1 {#3} { \l__hook_return_tl #2 } }
      { \prop_gput:Nnn #1 {#3} {#2} }
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_gput_code:nnn}
⟨latexrelease⟩                 {Providing~hooks}
⟨latexrelease⟩\cs_gset_protected:Npn \hook_gput_code:nnn #1 #2
⟨latexrelease⟩  { \__hook_normalize_hook_args:Nnn \__hook_gput_code:nnn {#1} {#2} }
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_gput_code:nnn #1 #2 #3
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_execute_immediately:nTF {#1}
⟨latexrelease⟩      {#3}
⟨latexrelease⟩      {
⟨latexrelease⟩        \__hook_if_usable:nTF {#1}
⟨latexrelease⟩          {
⟨latexrelease⟩            \__hook_hook_gput_code_do:nnn {#1} {#2} {#3}
⟨latexrelease⟩            \__hook_update_hook_code:n {#1}
⟨latexrelease⟩          }
⟨latexrelease⟩          {
⟨latexrelease⟩            \__hook_if_disabled:nTF {#1}
⟨latexrelease⟩              { \msg_error:nnn { hooks } { hook-disabled } {#1} }
⟨latexrelease⟩              { \__hook_try_declaring_generic_hook:nnn {#1} {#2} {#3} }
⟨latexrelease⟩          }
⟨latexrelease⟩      }
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_hook_gput_code_do:nnn #1 #2 #3
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_debug:n{\iow_term:x{****~ Add~ to~
⟨latexrelease⟩                       \__hook_if_usable:nF {#1} { undeclared~ }
⟨latexrelease⟩                       hook~ #1~ (#2)
⟨latexrelease⟩                       \on@line\space <-~ \tl_to_str:n{#3}} }
⟨latexrelease⟩    \str_if_eq:nnTF {#2} { top-level }
⟨latexrelease⟩      {
⟨latexrelease⟩        \str_if_eq:eeTF { top-level } { \__hook_currname_or_default: }
⟨latexrelease⟩          {
⟨latexrelease⟩            \__hook_init_structure:n {#1}
⟨latexrelease⟩            \__hook_tl_gput_right:cn { __hook_toplevel~#1 } {#3}
⟨latexrelease⟩          }
⟨latexrelease⟩          { \msg_error:nnn { hooks } { misused-top-level } {#1} }
⟨latexrelease⟩      }
⟨latexrelease⟩      {
⟨latexrelease⟩        \prop_get:cnNTF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
⟨latexrelease⟩          {
```

```
⟨latexrelease⟩             \prop_gput:cno { g__hook_#1_code_prop } {#2}
⟨latexrelease⟩               { \l__hook_return_tl #3 }
⟨latexrelease⟩           }
⟨latexrelease⟩           { \prop_gput:cnn { g__hook_#1_code_prop } {#2} {#3} }
⟨latexrelease⟩       }
⟨latexrelease⟩   }
⟨latexrelease⟩\cs_gset_protected:Npn \hook_gput_code_with_args:nnn #1#2#3 { }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_gput_code:nnn` *and others. These functions are documented on page 20.*)

`\__hook_chk_args_allowed:nn`

This macro checks if it is possible to add code with references to a hook's arguments for hook `#1`. It only does something if the function being run is `replacing_args`. This macro will error if the hook is declared and takes no arguments, then it will set `\__hook_replacing_args_false:` so that the macro which called it will add the code normally.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_chk_args_allowed:nn}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new_protected:Npn \__hook_chk_args_allowed:nn #1 #2
  {
    \__hook_if_replacing_args:TF
      {
        \__hook_if_declared:nT {#1}
          { \tl_if_empty:cT { c__hook_#1_parameter_tl } { \use_ii:nn } }
        \use_none:n
          {
            \msg_error:nnnn { hooks } { without-args } {#1} {#2}
            \__hook_replacing_args_false:
          }
      }
      { }
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_chk_args_allowed:nn}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_chk_args_allowed:nn
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_chk_args_allowed:nn`.)

`\__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. An implicitly declared hook doesn't have arguments (in principle), so use `\c_false_bool` here.

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

(*End of 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.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_try_declaring_generic_hook:nnn}
⟨latexrelease⟩                        {Hooks~with~args}
\cs_new_protected:Npn \__hook_try_declaring_generic_hook:nnn #1
  {
    \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
      \__hook_gput_code:nnn
      \__hook_gput_undeclared_hook:nnn
        {#1}
  }
\cs_new_protected:Npn \__hook_try_declaring_generic_next_hook:nn #1
  {
    \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
      \__hook_gput_next_code:nn
      \__hook_gput_next_do:nn
        {#1}
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_try_declaring_generic_hook:nnn}
⟨latexrelease⟩                        {Standardise~generic~hook~names}
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_try_declaring_generic_hook:nnn #1
⟨latexrelease⟩  {
```

```
⟨latexrelease⟩    \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
⟨latexrelease⟩      \hook_gput_code:nnn
⟨latexrelease⟩      \__hook_gput_undeclared_hook:nnn
⟨latexrelease⟩        {#1}
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_try_declaring_generic_next_hook:nn #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
⟨latexrelease⟩      \hook_gput_next_code:nn
⟨latexrelease⟩      \__hook_gput_next_do:nn
⟨latexrelease⟩        {#1}
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_try_declaring_generic_hook:nnn}
⟨latexrelease⟩                  {Standardise~generic~hook~names}
⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_declaring_generic_hook:nnn #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_try_declaring_generic_hook:nNNnn {#1}
⟨latexrelease⟩      \hook_gput_code:nnn \__hook_gput_undeclared_hook:nnn
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_declaring_generic_next_hook:nn #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_try_declaring_generic_hook:nNNnn {#1}
⟨latexrelease⟩      \hook_gput_next_code:nn \__hook_gput_next_do:nn
⟨latexrelease⟩  }
```

(*End of 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.

```
⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_declaring_generic_hook:nNNnn #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_file_hook:wTF #1 / \s__hook_mark
⟨latexrelease⟩      {
⟨latexrelease⟩        \exp_args:Ne \__hook_try_declaring_generic_hook_split:nNNnn
⟨latexrelease⟩          { \exp_args:Ne \__hook_file_hook_normalize:n {#1} }
⟨latexrelease⟩      }
⟨latexrelease⟩        { \__hook_try_declaring_generic_hook_split:nNNnn {#1} }
⟨latexrelease⟩   }
⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_declaring_generic_hook_split:nNNnn #1 #2 #3
⟨latexrelease⟩   {
⟨latexrelease⟩     \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
⟨latexrelease⟩        { #2 }
⟨latexrelease⟩        { #3 } {#1}
⟨latexrelease⟩   }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_try_declaring_generic_hook:nNNnn` *and* `\__hook_try_declaring_generic_hook_split:nNNnn`*.*)

`\__hook_try_declaring_generic_hook:wnTF`

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_try_declaring_generic_hook:wn}
⟨latexrelease⟩                      {Hooks~with~args}
\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
    #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
  {
    \__hook_if_generic:nTF {#5}
      {
        \__hook_if_usable:nF {#5}
          {
```

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.

```
            \str_if_eq:nnT {#1} { cmd }
              {
                \__hook_try_put_cmd_hook:n {#5}
                \__hook_make_usable:nn {#5} { 9 }
                \use_none:nnn
              }
```

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

Here we use `\__hook_make_usable:nn`, so that a `\hook_new:n` is still possible later. Generic hooks (except `cmd` hooks) take no arguments, so use zero as the second argument.

```
            \__hook_make_usable:nn {#5} { 0 }
```

```
          }
        \__hook_if_generic_reversed:nT {#5}
          { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
        \prg_return_true:
      }
      {
```

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:`.

```
        \__hook_if_deprecated_generic:nTF {#5}
          {
            \__hook_deprecated_generic_warn:n {#5}
            \exp_after:wN \__hook_declare_deprecated_generic:NNn
            \exp:w % \exp_end:
          }
          { \prg_return_false: }
      }
  }
```

__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).

```
\cs_new_protected:Npn \__hook_deprecated_generic_warn:n #1
  { \__hook_deprecated_generic_warn:w #1 \s__hook_mark }
\cs_new_protected:Npn \__hook_deprecated_generic_warn:w
    #1 / #2 / #3 \s__hook_mark
  {
    \if_cs_exist:w __hook~#1/#2/#3 \cs_end: \else:
      \msg_warning:nnnnn { hooks } { generic-deprecated } {#1} {#2} {#3}
    \fi:
    \cs_gset_eq:cN { __hook~#1/#2/#3 } \scan_stop:
  }
```

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.

__hook_do_deprecated_generic:Nn
__hook_do_deprecated_generic:Nw
__hook_declare_deprecated_generic:NNw
__hook_declare_deprecated_generic:NNw

```
\cs_new_protected:Npn \__hook_do_deprecated_generic:Nn #1 #2
  { \__hook_do_deprecated_generic:Nw #1 #2 \s__hook_mark }
\cs_new_protected:Npn \__hook_do_deprecated_generic:Nw #1
         #2 / #3 / #4 \s__hook_mark
  { #1 { #2 / #4 / #3 } }
\cs_new_protected:Npn \__hook_declare_deprecated_generic:NNn #1 #2 #3
  { \__hook_declare_deprecated_generic:NNw #1 #2 #3 \s__hook_mark }
\cs_new_protected:Npn \__hook_declare_deprecated_generic:NNw #1 #2
    #3 / #4 / #5 \s__hook_mark
  {
    \__hook_try_declaring_generic_hook:wnTF #3 / #5 / #4 / \scan_stop:
        { #3 / #5 / #4 }
      #1 #2 { #3 / #5 / #4 }
  }
⟨latexrelease⟩\EndIncludeInRelease

⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_try_declaring_generic_hook:wn}
⟨latexrelease⟩                  {Standardise~generic~hook~names}
⟨latexrelease⟩\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
⟨latexrelease⟩    #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_generic:nTF {#5}
⟨latexrelease⟩      {
⟨latexrelease⟩        \__hook_if_usable:nF {#5}
⟨latexrelease⟩          {
⟨latexrelease⟩            \str_if_eq:nnT {#1} { cmd }
⟨latexrelease⟩              { \__hook_try_put_cmd_hook:n {#5} }
⟨latexrelease⟩            \__hook_make_usable:n {#5}
⟨latexrelease⟩          }
⟨latexrelease⟩        \__hook_if_generic_reversed:nT {#5}
⟨latexrelease⟩          { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
⟨latexrelease⟩        \prg_return_true:
⟨latexrelease⟩      }
⟨latexrelease⟩      {
⟨latexrelease⟩        \__hook_if_deprecated_generic:nTF {#5}
⟨latexrelease⟩          {
⟨latexrelease⟩            \__hook_deprecated_generic_warn:n {#5}
⟨latexrelease⟩            \exp_after:wN \__hook_declare_deprecated_generic:NNn
⟨latexrelease⟩            \exp:w % \exp_end:
⟨latexrelease⟩          }
```

```
⟨latexrelease⟩            { \prg_return_false: }
⟨latexrelease⟩        }
⟨latexrelease⟩    }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2021/06/01}{\__hook_try_declaring_generic_hook:wn}
⟨latexrelease⟩                 {Support~cmd~hooks}
⟨latexrelease⟩\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
⟨latexrelease⟩    #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
⟨latexrelease⟩  {
⟨latexrelease⟩    \tl_if_empty:nTF {#2}
⟨latexrelease⟩      { \prg_return_false: }
⟨latexrelease⟩      {
⟨latexrelease⟩        \prop_if_in:NnTF \c__hook_generics_prop {#1}
⟨latexrelease⟩          {
⟨latexrelease⟩            \__hook_if_usable:nF {#5}
⟨latexrelease⟩              {
⟨latexrelease⟩                \str_if_eq:nnT {#1} { cmd }
⟨latexrelease⟩                  { \__hook_try_put_cmd_hook:n {#5} }
⟨latexrelease⟩                \__hook_make_usable:n {#5}
⟨latexrelease⟩              }
⟨latexrelease⟩            \prop_if_in:NnTF \c__hook_generics_reversed_ii_prop {#2}
⟨latexrelease⟩              { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
⟨latexrelease⟩              {
⟨latexrelease⟩                \prop_if_in:NnT \c__hook_generics_reversed_iii_prop {#3}
⟨latexrelease⟩                  { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
⟨latexrelease⟩              }
⟨latexrelease⟩            \prg_return_true:
⟨latexrelease⟩          }
⟨latexrelease⟩          { \prg_return_false: }
⟨latexrelease⟩      }
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_try_declaring_generic_hook:wn}
⟨latexrelease⟩                 {Support~cmd~hooks}
⟨latexrelease⟩\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
⟨latexrelease⟩    #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
⟨latexrelease⟩  {
⟨latexrelease⟩    \tl_if_empty:nTF {#2}
⟨latexrelease⟩      { \prg_return_false: }
⟨latexrelease⟩      {
⟨latexrelease⟩        \prop_if_in:NnTF \c__hook_generics_prop {#1}
⟨latexrelease⟩          {
```

```
⟨latexrelease⟩             \__hook_if_declared:nF {#5} { \hook_new:n {#5} }
⟨latexrelease⟩             \prop_if_in:NnTF \c__hook_generics_reversed_ii_prop {#2}
⟨latexrelease⟩               { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
⟨latexrelease⟩               {
⟨latexrelease⟩                 \prop_if_in:NnT \c__hook_generics_reversed_iii_prop {#3}
⟨latexrelease⟩                   { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
⟨latexrelease⟩               }
⟨latexrelease⟩             \prg_return_true:
⟨latexrelease⟩           }
⟨latexrelease⟩           { \prg_return_false: }
⟨latexrelease⟩       }
⟨latexrelease⟩   }
⟨latexrelease⟩\EndIncludeInRelease
```

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

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

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

```
⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_if_file_hook:w}
⟨latexrelease⟩                 {Standardise~generic~hook~names}
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_if_file_hook:w}
⟨latexrelease⟩                 {Standardise~generic~hook~names}
⟨latexrelease⟩\prg_new_conditional:Npnn \__hook_if_file_hook:w
⟨latexrelease⟩    #1 / #2 / #3 \s__hook_mark { TF }
⟨latexrelease⟩  {
⟨latexrelease⟩    \str_if_eq:nnTF {#1} { file }
⟨latexrelease⟩      {
⟨latexrelease⟩        \bool_lazy_or:nnTF
⟨latexrelease⟩            { \tl_if_empty_p:n {#3} }
⟨latexrelease⟩            { \str_if_eq_p:nn {#3} { / } }
⟨latexrelease⟩          { \prg_return_false: }
⟨latexrelease⟩          {
⟨latexrelease⟩            \prop_if_in:NnTF \c__hook_generics_file_prop {#2}
⟨latexrelease⟩              { \prg_return_true: }
⟨latexrelease⟩              { \prg_return_false: }
⟨latexrelease⟩          }
⟨latexrelease⟩      }
⟨latexrelease⟩      { \prg_return_false: }
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
```

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

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

```
⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_file_hook_normalize:n}
⟨latexrelease⟩                            {Standardise~generic~hook~names}
⟨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}`.

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

```
⟨latexrelease⟩\cs_new:Npn \__hook_strip_double_slash:w #1/#2/#3 // #4 \s__hook_mark
⟨latexrelease⟩  {
⟨latexrelease⟩    \tl_if_empty:nTF {#4}
⟨latexrelease⟩      { #1/#2/#3 }
⟨latexrelease⟩      { \__hook_strip_double_slash:w #1/#2/#3 / #4 \s__hook_mark }
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
```

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

\\c__hook_generic_cmd/./before_tl
\\c__hook_generic_cmd/./after_tl
\\c__hook_generic_env/./before_tl
\\c__hook_generic_env/./after_tl
\\c__hook_generic_file/./before_tl
\\c__hook_generic_file/./after_tl
\\c__hook_generic_package/./before_tl
\\c__hook_generic_package/./after_tl
\\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

Token lists defining the possible generic hooks. We don't provide any user interface to this as this is meant to be static.

cmd The generic hooks used for commands.

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

file, package, class, include The generic hooks used when loading a file

```
⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\c__hook_generics_prop}
⟨latexrelease⟩                  {Standardise~generic~hook~names}
\clist_map_inline:nn { cmd , env , file , package , class , include }
  {
    \tl_const:cn { c__hook_generic_#1/./before_tl } { + }
    \tl_const:cn { c__hook_generic_#1/./after_tl  } { - }
  }
\tl_const:cn { c__hook_generic_env/./begin_tl } { + }
\tl_const:cn { c__hook_generic_env/./end_tl   } { + }

\tl_const:cn { c__hook_generic_include/./end_tl } { - }
\tl_const:cn { c__hook_generic_include/./excluded_tl } { + }
```

Deprecated generic hooks:

```
\clist_map_inline:nn { file , package , class , include }
  {
    \tl_const:cn { c__hook_deprecated_#1/./before_tl } { }
    \tl_const:cn { c__hook_deprecated_#1/./after_tl  } { }
  }
\tl_const:cn { c__hook_deprecated_include/./end_tl } { }
⟨latexrelease⟩\EndIncludeInRelease

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

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

\c__hook_generics_reversed_ii_prop
\c__hook_generics_reversed_iii_prop
\c__hook_generics_file_prop

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

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

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

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

\c__hook_parameter_cmd/./before_tl
\c__hook_parameter_cmd/./after_tl

Token lists defining the number of arguments for a given type of generic hook.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\c__hook_parameter_cmd/./before_tl}
⟨latexrelease⟩                          {Hooks~with~args}
```

cmd hooks are declared with 9 arguments because they have a variable number of arguments (depending on the command they are attached to), so we use the maximum here.

```
\tl_const:cn { c__hook_parameter_cmd/./before_tl } { #1#2#3#4#5#6#7#8#9 }
\tl_const:cn { c__hook_parameter_cmd/./after_tl }  { #1#2#3#4#5#6#7#8#9 }
```

```
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\c__hook_parameter_cmd/./before_tl}
⟨latexrelease⟩                          {Hooks~with~args}
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* \c__hook_parameter_cmd/./before_tl *and* \c__hook_parameter_cmd/./after_tl.)

\hook_gremove_code:nn
__hook_gremove_code:nn

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

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_gremove_code:nn}
⟨latexrelease⟩                          {Hooks~with~args}
\cs_new_protected:Npn \hook_gremove_code:nn #1 #2
  { \__hook_normalize_hook_args:Nnn \__hook_gremove_code:nn {#1} {#2} }
\cs_new_protected:Npn \__hook_gremove_code:nn #1 #2
  {
```

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).

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

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

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*⟩.

```
        \str_if_eq:nnTF {#2} {*}
          {
            \prop_gclear:c { g__hook_#1_code_prop }
            \__hook_toplevel_gset:nn {#1} { }
            \__hook_next_gset:nn {#1} { }
          }
          {
```

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

```
          \str_if_eq:nnTF {#2} { top-level }
            { \__hook_toplevel_gset:nn {#1} { } }
            {
              \prop_gpop:cnNF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
                { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
            }
        }
```

Finally update the code, if the hook exists.

```
      \__hook_if_usable:nT {#1}
        { \__hook_update_hook_code:n {#1} }
    }
```

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

```
    {
      \__hook_if_deprecated_generic:nTF {#1}
        {
          \__hook_deprecated_generic_warn:n {#1}
          \__hook_do_deprecated_generic:Nn \__hook_gremove_code:nn {#1} {#2}
        }
        { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
    }
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_gremove_code:nn}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_new_protected:Npn \__hook_gremove_code:nn #1 #2
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_structure_exist:nTF {#1}
⟨latexrelease⟩      {
⟨latexrelease⟩        \str_if_eq:nnTF {#2} {*}
⟨latexrelease⟩          {
⟨latexrelease⟩            \prop_gclear:c { g__hook_#1_code_prop }
⟨latexrelease⟩            \__hook_tl_gclear:c { __hook_toplevel~#1 }
⟨latexrelease⟩            \__hook_tl_gclear:c { __hook_next~#1 }
⟨latexrelease⟩          }
⟨latexrelease⟩          {
⟨latexrelease⟩            \str_if_eq:nnTF {#2} { top-level }
⟨latexrelease⟩              { \__hook_tl_gclear:c { __hook_toplevel~#1 } }
⟨latexrelease⟩              {
⟨latexrelease⟩                \prop_gpop:cnNF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
⟨latexrelease⟩                    { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
⟨latexrelease⟩                }
⟨latexrelease⟩            }
⟨latexrelease⟩          \__hook_if_usable:nT {#1}
⟨latexrelease⟩            { \__hook_update_hook_code:n {#1} }
⟨latexrelease⟩        }
⟨latexrelease⟩        {
⟨latexrelease⟩          \__hook_if_deprecated_generic:nTF {#1}
⟨latexrelease⟩            {
⟨latexrelease⟩              \__hook_deprecated_generic_warn:n {#1}
⟨latexrelease⟩              \__hook_do_deprecated_generic:Nn \__hook_gremove_code:nn {#1} {#2}
⟨latexrelease⟩            }
⟨latexrelease⟩            { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
⟨latexrelease⟩        }
⟨latexrelease⟩    }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_gremove_code:nn` *and* `\__hook_gremove_code:nn`. *This function is documented on page 21.*)

`\__hook_cs_gput_right:nnn`
`\__hook_cs_gput_right_fast:nnn`
`\__hook_cs_gput_right_slow:nnn`
`\__hook_code_gset_auxi:nnnn`
`\__hook_code_gset_auxi:eeen`

This macro is used to append code to the `toplevel` and `next` token lists, trating them correctly depending on their number of arguments, and depending if the code being added should have parameter tokens understood as parameters, or doubled to be stored as parameter tokens.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_cs_gput_right:nnn}
⟨latexrelease⟩                    {Hooks~with~args}
```

Check if the current hook is declared and takes no arguments. In this case, we short-circuit and use the simpler and much faster approach that doesn't require hash-doubling.

```
\cs_new_protected:Npn \__hook_cs_gput_right:nnn #1 #2
  {
    \if:w T
        \__hook_if_declared:nF {#2} { F }
        \tl_if_empty:cF { c__hook_#2_parameter_tl } { F }
          T
      \exp_after:wN \__hook_cs_gput_right_fast:nnn
    \else:
      \exp_after:wN \__hook_cs_gput_right_slow:nnn
    \fi:
        {#1} {#2}
  }
\cs_new_protected:Npn \__hook_cs_gput_right_fast:nnn #1 #2 #3
```

```
  { \cs_gset:cpx { __hook#1~#2 } { \exp_not:v { __hook#1~#2 } \exp_not:n {#3} } }
\cs_new_protected:Npn \__hook_cs_gput_right_slow:nnn #1 #2 #3
  {
```

The auxiliary `\__hook_code_gset_auxi:eeen` just does the assignment at the end. Its first argument is the parameter text of the macro, which is chosen here depending if `\c__hook_`⟨*hook*⟩`_parameter_tl` exists, if the hook is declared, and if it's a generic hook.

```
    \cs_if_exist:cF { __hook#1~#2 }
      { \__hook_code_gset_aux:nnn {#1} {#2} { } }
    \__hook_code_gset_auxi:eeen
      {
        \__hook_if_declared:nTF {#2}
          { \tl_use:c { c__hook_#2_parameter_tl } }
          {
            \__hook_if_generic:nTF {#2}
              { \__hook_generic_parameter:n {#2} }
              { \c__hook_nine_parameters_tl }
          }
      }
```

Here we take the existing code in the macro, expand it with as many arguments as it takes, then double the hashes so the code can be reused.

PhO: Maybe can be improved. The case of adding to an empty cs can be optimised by quickly checking `\cs_replacement_spec`.

```
      {
        \exp_args:NNo \exp_args:No \__hook_double_hashes:n
          {
            \cs:w __hook#1~#2 \exp_last_unbraced:Ne \cs_end:
              { \__hook_braced_cs_parameter:n { __hook#1~#2 } }
          }
      }
```

Now the new code: if we are replacing arguments, then hashes are left untouched, otherwise they are doubled.

```
      {
        \__hook_if_replacing_args:TF
          { \exp_not:n }
          { \__hook_double_hashes:n }
            {#3}
      }
```

And finally, the csname which we'll define with all the above.

```
      { __hook#1~#2 }
  }
```

And as promised, the auxiliary that does the definition.

```
\cs_new_protected:Npn \__hook_code_gset_auxi:nnnn #1 #2 #3 #4
  { \cs_gset:cpn {#4} #1 { #2 #3 } }
\cs_generate_variant:Nn \__hook_code_gset_auxi:nnnn { eeen }
```

```
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_cs_gput_right:nnn}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_cs_gput_right:nnn
⟨latexrelease⟩\cs_undefine:N \__hook_cs_gput_right_fast:nnn
⟨latexrelease⟩\cs_undefine:N \__hook_cs_gput_right_slow:nnn
⟨latexrelease⟩\cs_undefine:N \__hook_code_gset_auxi:nnnn
⟨latexrelease⟩\EndIncludeInRelease
```

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

`\__hook_code_gset:nn`
`\__hook_code_gset:ne`
`\__hook_toplevel_gset:nn`
`\__hook_next_gset:nn`
`\__hook_code_gset_aux:nnn`

These macros define `\__hook`⟨*type*⟩␣⟨*hook*⟩ (with ⟨*type*⟩ being `_next`, `_toplevel`, or empty) with the given code and the parameters stored in `\c__hook_`⟨*hook*⟩`_parameter_tl` (or none, if that doesn't exist).

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_code_gset:nn}
⟨latexrelease⟩                 {Hooks~with~args}
\cs_new_protected:Npn \__hook_code_gset:nn
  { \__hook_code_gset_aux:nnn { } }
\cs_new_protected:Npn \__hook_toplevel_gset:nn
  { \__hook_code_gset_aux:nnn { _toplevel } }
\cs_new_protected:Npn \__hook_next_gset:nn
  { \__hook_code_gset_aux:nnn { _next } }
\cs_new_protected:Npn \__hook_code_gset_aux:nnn #1 #2 #3
  {
    \cs_gset:cpn { __hook#1~#2 \exp_last_unbraced:Ne }
      { \__hook_parameter:n {#2} }
      {#3}
  }
\cs_generate_variant:Nn \__hook_code_gset:nn { ne }
```

```
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_code_gset:nn}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_code_gset:nn
⟨latexrelease⟩\cs_undefine:N \__hook_toplevel_gset:nn
⟨latexrelease⟩\cs_undefine:N \__hook_next_gset:nn
⟨latexrelease⟩\cs_undefine:N \__hook_code_gset_aux:nnn
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* __hook_code_gset:nn *and others.*)

__hook_normalise_cs_args:nn

This macro normalises the parameters of the macros __hook⟨*type*⟩␣⟨*hook*⟩ to take the right number of arguments after a hook is declared. At this point we know \c__hook_⟨*hook*⟩_parameter_tl exists, so use that to count the arguments and use that as ⟨*parameter text*⟩ for the newly (re)defined macro.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_normalise_cs_args:nn}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new_protected:Npn \__hook_normalise_cs_args:nn #1 #2
  {
    \cs_if_exist:cT { __hook#1~#2 }
      {
        \__hook_code_gset_auxi:eeen
          { \tl_use:c { c__hook_#2_parameter_tl } }
          {
            \exp_args:NNo \exp_args:No \__hook_double_hashes:n
              {
                \cs:w __hook#1~#2 \exp_last_unbraced:Ne \cs_end:
                  { \__hook_braced_cs_parameter:n { __hook#1~#2 } }
              }
          }
          { }
          { __hook#1~#2 }
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_normalise_cs_args:nn}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_normalise_cs_args:nn
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* __hook_normalise_cs_args:nn*.*)

__hook_normalise_code_pool:n
__hook_set_normalise_fn:nn

This one's a bit of a hack. It takes a hook, and iterates over its code pool (\g_-_hook_⟨*hook*⟩_code_prop), redefining each code label to use only valid arguments. This is used when, for example, a code is added referencing arguments #1 and #2, but the hook has only #1. In this example, every reference to #2 is changed to ##2. This is done because otherwise TEX will throw a low-level error every time some change happens to the hook (code is added, a rule is set, etc), which can get quite repetitive for no good reason.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_normalise_code_pool:n}
```

```
⟨latexrelease⟩                    {Hooks~with~args}
\cs_new_protected:Npn \__hook_normalise_code_pool:n #1
  {
```

First, call `\__hook_set_normalise_fn:nn` with the hook name to set everything up, then we'll loop over the hook's code pool applying the normalisation above. After that's done, copy the temporary property list back to the hook's.

```
    \__hook_set_normalise_fn:nn {#1} { Offending~label:~'##1' }
    \prop_clear:N \l__hook_work_prop
    \prop_map_function:cN { g__hook_#1_code_prop } \__hook_normalise_fn:nn
    \prop_gset_eq:cN { g__hook_#1_code_prop } \l__hook_work_prop
  }
```

The sole purpose of this function is to define `\__hook_normalise_fn:nn`, which will then do the correcting of the code being added to the hook.

```
\cs_new_protected:Npn \__hook_set_normalise_fn:nn #1 #2
  {
```

To start, we define two auxiliary token lists. `\l__hook_tmpb_tl` contains:

```
{\c__hook_hashes_tl 1}
{\c__hook_hashes_tl 2}
...
{\c__hook_hashes_tl 9}
```

```
    \cs_set:Npn \__hook_tmp:w ##1##2##3##4##5##6##7##8##9 { }
    \tl_set:Ne \l__hook_tmpb_tl
      { \__hook_braced_cs_parameter:n { __hook_tmp:w } }
    \group_begin:
      \__hook_tl_set:cn { c__hook_hash_tl } { \exp_not:N \c__hook_hashes_tl }
      \use:e
        {
    \group_end:
    \tl_set:Nn \exp_not:N \l__hook_tmpb_tl { \l__hook_tmpb_tl }
        }
```

And `\l__hook_tmpa_tl` contains:

```
{\c__hook_hash_tl 1}
{\c__hook_hash_tl 2}
...
{\c__hook_hash_tl <n>}
```

with ⟨*n*⟩ being the number of arguments declared for the hook.

```
    \exp_last_unbraced:NNf
    \cs_set:Npn \__hook_tmp:w { \__hook_parameter:n {#1} } { }
    \tl_set:Ne \l__hook_tmpa_tl { \__hook_braced_cs_parameter:n { __hook_tmp:w } }
```

Now this function does the fun part. It is meant to be used with `\prop_map_-function:NN`, taking a label name in `##1` and the code stored in that label in `##2`.

```
    \cs_gset_protected:Npx \__hook_normalise_fn:nn ##1 ##2
      {
```

Here we'll define two auxiliary macros: the first one throws an error when it detects an invalid argument reference. It piggybacks on TEX's low-level "Illegal parameter number" error, but it defines a weirdly-named control sequence so that the error comes out nicely formatted. For example, if the label "badpkg" adds some code that references argument `#3` in the hook "foo", which takes only two arguments, the error will be:

```
! Illegal parameter number in definition of hook 'foo'.
(hooks)              Offending label: 'badpkg'.
<to be read again>
                    3
```

At the point of this definition, the error is raised if the code happens to reference an invalid argument. If it was possible to detect that this definition raised no error, the next step would be unnecessary. We'll do all this in a group so this weird definition doesn't leak out, and set `\tex_escapechar:D` to -1 so this hack shows up extra nice in the case of an error.

```
        \group_begin:
          \int_set:Nn \tex_escapechar:D { -1 }
          \cs_set:cpn
              {
                hook~'#1'. ^^J
                (hooks) \prg_replicate:nn { 13 } { ~ }
                #2 % more message text
              }
              \exp_not:v { c__hook_#1_parameter_tl }
            {##2}
        \group_end:
```

This next macro, with a much less fabulous name, takes always nine arguments, and it just transfers the code `##2` under the label `##1` to the temporary property list. The first ⟨*n*⟩ arguments are taken from `\l__hook_tmpa_tl`, and the other $9 - \langle n \rangle$

taken from `\l__hook_tmpb_tl` (which contains twice as many `#` tokens as the former). Then, `\__hook_double_hashes:n` is used to double non-argument hashes, and expand the `\c__hook_hash_tl` and `\c__hook_hashes_tl` to the actual parameter tokens.

```
          \cs_set:Npn \exp_not:N \__hook_tmp:w
              \exp_not:V \c__hook_nine_parameters_tl
            {
              \prop_put:Nne \exp_not:N \l__hook_work_prop
                {##1} { \exp_not:N \__hook_double_hashes:n {##2} }
            }
```

This next macro, with a much less fabulous name, takes always nine arguments, and it just transfers the code `##2` under the label `##1` to the temporary property list. The first ⟨n⟩ arguments are taken from `\l__hook_tmpa_tl`, and the other $9 - \langle n\rangle$ taken from `\l__hook_tmpb_tl` (which contains twice as many `#` tokens as the former). Then, `\__hook_double_hashes:n` is used to double non-argument hashes, and expand the `\c__hook_hash_tl` and `\c__hook_hashes_tl` to the actual parameter tokens.

```
          \exp_not:N \__hook_tmp:w
            \exp_not:V \l__hook_tmpa_tl
            \exp_args:No \exp_not:o
              { \exp_after:wN \__hook_tmp:w \l__hook_tmpb_tl }
        }
    }
\cs_new_eq:NN \__hook_normalise_fn:nn ?
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_normalise_code_pool:n}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_normalise_code_pool:n
⟨latexrelease⟩\EndIncludeInRelease
```

`\__hook_cs_if_empty_p:c`
`\__hook_cs_if_empty:cTF`

Check if the expansion of a control sequence is empty by looking at its replacement text.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_cs_if_empty:c}
⟨latexrelease⟩                  {Hooks~with~args}
\prg_new_conditional:Npnn \__hook_cs_if_empty:c #1 { p, T, F, TF }
  {
    \if:w \scan_stop: \__hook_replacement_spec:c {#1} \scan_stop:
      \prg_return_true:
    \else:
      \prg_return_false:
```

```
    \fi:
  }
\cs_new:Npn \__hook_replacement_spec:c #1
  {
    \exp_args:Nc \token_if_macro:NT {#1}
      { \cs_replacement_spec:c {#1} }
  }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_cs_if_empty:c}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_cs_if_empty:c
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_normalise_code_pool:n`, `\__hook_set_normalise_fn:nn`, *and* `\__hook_cs_if_-empty:cTF`.)

`\__hook_braced_cs_parameter:n`
`\__hook_braced_hidden_loop:w`
`\__hook_cs_parameter_count:N`
`\__hook_cs_parameter_count:w`
`\__hook_cs_end:w`

Looks at the ⟨*parameter text*⟩ of a control sequence, and returns a run of "hidden" braced parameters for that macro. This works as long as the macros take a simple run of zero to nine arguments. The parameters are "hidden" because the parameter tokens are returned inside `\c__hook_hash_tl` instead of explicitly, so that `\__hook_double_hashes:n` won't touch these.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_braced_cs_parameter:n}
⟨latexrelease⟩                 {Hooks~with~args}
\cs_new:Npn \__hook_braced_cs_parameter:n #1
  {
    \exp_last_unbraced:Ne \__hook_braced_hidden_loop:w
      { \exp_args:Nc \__hook_cs_parameter_count:N {#1} } ? \s__hook_mark
  }
\cs_new:Npn \__hook_braced_hidden_loop:w #1
  {
    \if:w ? #1
      \__hook_use_i_delimit_by_s_mark:nw
    \fi:
    { \exp_not:N \c__hook_hash_tl #1 }
    \__hook_braced_hidden_loop:w
  }
\cs_new:Npn \__hook_cs_parameter_count:N #1
  {
    \exp_last_unbraced:Nf \__hook_cs_parameter_count:w
      { \token_if_macro:NT #1 { \cs_parameter_spec:N #1 } }
      ? \__hook_cs_end:w ? \__hook_cs_end:w ? \__hook_cs_end:w
      ? \__hook_cs_end:w ? \__hook_cs_end:w ? \__hook_cs_end:w
      ? \__hook_cs_end:w ? \__hook_cs_end:w ? \__hook_cs_end:w
      \s__hook_mark
  }
\cs_new:Npn \__hook_cs_parameter_count:w #1#2 #3#4 #5#6 #7#8
  { #2 #4 #6 #8 \__hook_cs_parameter_count:w }
\cs_new:Npn \__hook_cs_end:w #1 \s__hook_mark { }
⟨latexrelease⟩\EndIncludeInRelease
```

This function can't be undefined when rolling back because it's used at the end of this module to adequate the hook data structures to previous versions.

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_braced_cs_parameter:n}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\EndIncludeInRelease
```

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

`\__hook_braced_parameter:n`
`\__hook_braced_real_loop:w`

This one is used in simpler cases, where no special handling of hashes is required. This is used only inside `\__hook_initialize_hook_code:n`, so it assumes `\c__hook_⟨hook⟩_parameter_tl` is defined, but should work otherwise.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_braced_parameter:n}
⟨latexrelease⟩                 {Hooks~with~args}
\cs_new:Npn \__hook_braced_parameter:n #1
  {
    \if_case:w
      \int_eval:n
        { \exp_args:Nv \str_count:n { c__hook_#1_parameter_tl } / 3 }
      \exp_stop_f:
    \or: {##1}
    \or: {##1} {##2}
    \or: {##1} {##2} {##3}
    \or: {##1} {##2} {##3} {##4}
    \or: {##1} {##2} {##3} {##4} {##5}
    \or: {##1} {##2} {##3} {##4} {##5} {##6}
    \or: {##1} {##2} {##3} {##4} {##5} {##6} {##7}
    \or: {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8}
    \or: {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8} {##9}
    \else:
      \msg_expandable_error:nnn { latex2e } { should-not-happen }
        { Invalid~parameter~spec. }
    \fi:
  }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_braced_parameter:n}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_braced_parameter:n
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_braced_parameter:n` *and* `\__hook_braced_real_loop:w`.)

`\__hook_parameter:n`

This is just a shortcut to `e`- or `f`-expand to the ⟨*parameter text*⟩ of the hook.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_parameter:n}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new:Npn \__hook_parameter:n #1
  {
    \cs:w c__hook_
    \tl_if_exist:cTF { c__hook_#1_parameter_tl }
      { #1_parameter } { empty }
    _tl \cs_end:
  }
\cs_new:Npn \__hook_generic_parameter:n #1
  { \__hook_generic_parameter:w #1 / / / \s__hook_mark }
\cs_new:Npn \__hook_generic_parameter:w #1 / #2 / #3 / #4 \s__hook_mark
  {
    \cs_if_exist_use:cF { c__hook_parameter_#1/./#3_tl }
      { \c__hook_empty_tl }
  }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_parameter:n}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_parameter:n
⟨latexrelease⟩\cs_undefine:N \__hook_generic_parameter:n
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_parameter:n`.)

4.7 Setting rules for hooks code

`\g__hook_??_code_prop`
`\__hook~??`
`\g__hook_??_reversed_tl`
`\c__hook_??_parameter_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.

```
\prop_new:c { g__hook_??_code_prop }
\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.

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

The parameter text for the "default" hook is empty.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\c__hook_??_parameter_tl}
⟨latexrelease⟩                 {Hooks~with~args}
\tl_const:cn { c__hook_??_parameter_tl } { }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\c__hook_??_parameter_tl}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:c { c__hook_??_parameter_tl }
⟨latexrelease⟩\EndIncludeInRelease
```

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

`\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).

```
\cs_new_protected:Npn \hook_gset_rule:nnnn #1#2#3#4
  {
    \__hook_normalize_hook_rule_args:Nnnnn \__hook_gset_rule:nnnn
      {#1} {#2} {#3} {#4}
  }
⟨latexrelease⟩\IncludeInRelease{2022/06/01}{\__hook_gset_rule:nnnn}
⟨latexrelease⟩                 {Refuse~setting~rule~for~one-time~hooks}
\cs_new_protected:Npn \__hook_gset_rule:nnnn #1#2#3#4
  {
    \__hook_if_deprecated_generic:nT {#1}
      {
        \__hook_deprecated_generic_warn:n {#1}
        \__hook_do_deprecated_generic:Nn \__hook_gset_rule:nnnn {#1}
          {#2} {#3} {#4}
        \__hook_use_none_delimit_by_s_mark:w
      }
    \__hook_if_execute_immediately:nT {#1}
      {
        \msg_error:nnnnnn { hooks } { rule-too-late }
          {#1} {#2} {#3} {#4}
        \__hook_use_none_delimit_by_s_mark:w
      }
```

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

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

Then we clear any previous relationship between both labels.

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

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

```
  \cs_if_exist_use:cTF { __hook_rule_#3_gset:nnn }
    {
        {#1} {#2} {#4}
      \__hook_update_hook_code:n {#1}
    }
    {
      \msg_error:nnnnnn { hooks } { unknown-rule }
        {#1} {#2} {#3} {#4}
    }
  \s__hook_mark
}
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_gset_rule:nnnn}
⟨latexrelease⟩                 {Refuse~setting~rule~for~one-time~hooks}
⟨latexrelease⟩\cs_new_protected:Npn \__hook_gset_rule:nnnn #1#2#3#4
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_deprecated_generic:nT {#1}
⟨latexrelease⟩      {
⟨latexrelease⟩        \__hook_deprecated_generic_warn:n {#1}
⟨latexrelease⟩        \__hook_do_deprecated_generic:Nn \__hook_gset_rule:nnnn {#1}
⟨latexrelease⟩          {#2} {#3} {#4}
⟨latexrelease⟩        \exp_after:wN \use_none:nnnnnnnnn \use_none:n
⟨latexrelease⟩      }
⟨latexrelease⟩    \__hook_init_structure:n {#1}
⟨latexrelease⟩    \__hook_rule_gclear:nnn {#1} {#2} {#4}
⟨latexrelease⟩    \cs_if_exist_use:cTF { __hook_rule_#3_gset:nnn }
⟨latexrelease⟩      {
```

```
⟨latexrelease⟩            {#1} {#2} {#4}
⟨latexrelease⟩          \__hook_update_hook_code:n {#1}
⟨latexrelease⟩        }
⟨latexrelease⟩        {
⟨latexrelease⟩          \msg_error:nnnnnn { hooks } { unknown-rule }
⟨latexrelease⟩            {#1} {#2} {#3} {#4}
⟨latexrelease⟩        }
⟨latexrelease⟩    }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_gset_rule:nnnn` *and* `\__hook_gset_rule:nnnn`. *This function is documented on page 21.*)

`\__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.

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

(*End of 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`.

```
\cs_new_protected:Npn \__hook_rule_voids_gset:nnn #1#2#3
```

```
  {
    \__hook_tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
      { \__hook_label_ordered:nnTF {#2} {#3} { -> } { <- } }
  }
```

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

`\__hook_rule_incompatible-error_gset:nnn`
`\__hook_rule_incompatible-warning_gset:nnn`

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

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

(*End of definition for* `\__hook_rule_incompatible-error_gset:nnn` *and* `\__hook_rule_incompatible-warning_-gset:nnn`.)

`\__hook_rule_unrelated_gset:nnn`
`\__hook_rule_gclear:nnn`

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

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

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

`\__hook_label_pair:nn`

Ensure that the lexically greater label comes first.

```
\cs_new:Npn \__hook_label_pair:nn #1#2
  {
    \if_case:w \__hook_str_compare:nn {#1} {#2} \exp_stop_f:
           #1 | #1 %  0
    \or:   #1 | #2 % +1
    \else: #2 | #1 % -1
    \fi:
  }
```

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

`\__hook_label_ordered_p:nn`
`\__hook_label_ordered:nn`***TF***

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

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

(*End of 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.

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

(*End of 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.

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

(*End of 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.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_initialize_all:}
⟨latexrelease⟩                        {Hooks~with~args}
\cs_new_protected:Npn \__hook_initialize_all:
  {
```

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

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

Now we loop over all hooks that have been defined and update each of them. Here we have to determine if the hook has arguments so that auxiliaries know what to do with hashes. We look at `\c__hook_⟨hook⟩_parameter_tl`, if it has any parameters, and set `replacing_args` accordingly.

```
    \__hook_debug:n { \prop_gclear:N \g__hook_used_prop }
    \seq_map_inline:Nn \g__hook_all_seq
      {
        \tl_if_empty:cTF { c__hook_##1_parameter_tl }
          { \__hook_replacing_args_false: }
          { \__hook_replacing_args_true: }
        \__hook_update_hook_code:n {##1}
        \__hook_replacing_args_reset:
      }
```

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

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

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.

```
    \__hook_post_initialization_defs:
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_initialize_all:}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_initialize_all:
⟨latexrelease⟩  {
⟨latexrelease⟩    \cs_gset_eq:NN \__hook_update_hook_code:n \__hook_initialize_hook_code:n
⟨latexrelease⟩    \__hook_debug:n { \prop_gclear:N \g__hook_used_prop }
⟨latexrelease⟩    \seq_map_inline:Nn \g__hook_all_seq
⟨latexrelease⟩      { \__hook_update_hook_code:n {##1} }
⟨latexrelease⟩    \__hook_debug:n
⟨latexrelease⟩      {
⟨latexrelease⟩        \iow_term:x{^^JAll~ initialized~ (non-empty)~ hooks:}
⟨latexrelease⟩        \prop_map_inline:Nn \g__hook_used_prop
⟨latexrelease⟩          {
⟨latexrelease⟩            \iow_term:x
⟨latexrelease⟩                 { ^^J ~ ##1 ~ -> ~ \cs_replacement_spec:c { __hook~##1 } ~ }
⟨latexrelease⟩             }
⟨latexrelease⟩         }
⟨latexrelease⟩     \cs_gset_eq:NN \hook_use:n \__hook_use_initialized:n
⟨latexrelease⟩     \cs_gset_eq:NN \__hook_preamble_hook:n \use_none:n
⟨latexrelease⟩   }
⟨@@=⟩
⟨latexrelease⟩\cs_gset_eq:NN \@expl@@@initialize@all@@
⟨latexrelease⟩                 \__hook_initialize_all:
⟨@@=hook⟩
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of 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.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_initialize_hook_code:n}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new_protected:Npn \__hook_initialize_hook_code:n #1
  {
    \__hook_debug:n
      { \iow_term:x { ^^J Update~code~for~hook~'#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.

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

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.

```
    \__hook_if_usable:nT {#1}
      {
        \prop_if_empty:cTF { g__hook_#1_code_prop }
          {
            \__hook_code_gset:ne {#1}
              {
```

The hook may take arguments, so we add a run of braced parameters after the `_next` and `_toplevel` macros, so that the arguments passed to the hook are forwarded to them.

```
              \exp_not:c { __hook_toplevel~#1 } \__hook_braced_parameter:n {#1}
              \exp_not:c { __hook_next~#1 } \__hook_braced_parameter:n {#1}
            }
          }
          {
```

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.

```
            \__hook_if_reversed:nTF {#1}
              { \cs_set_eq:NN \__hook_tl_gput:Nn    \__hook_tl_gput_left:Nn
                \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_left:NV  }
              { \cs_set_eq:NN \__hook_tl_gput:Nn    \__hook_tl_gput_right:Nn
                \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.

```
            \prop_set_eq:Nc \l__hook_work_prop { g__hook_#1_code_prop }
            \__hook_initialize_single:ccn
              { __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.

```
            \__hook_debug:n
              { \exp_args:NNx \prop_gput:Nnn \g__hook_used_prop {#1} { } }
          }
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_initialize_hook_code:n}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_initialize_hook_code:n #1
```

```
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_debug:n
⟨latexrelease⟩      { \iow_term:x { ^^J Update~code~for~hook~'#1' \on@line :^^J } }
⟨latexrelease⟩    \__hook_include_legacy_code_chunk:n {#1}
⟨latexrelease⟩    \__hook_if_usable:nT {#1}
⟨latexrelease⟩      {
⟨latexrelease⟩        \prop_if_empty:cTF { g__hook_#1_code_prop }
⟨latexrelease⟩          {
⟨latexrelease⟩            \__hook_tl_gset:co { __hook~#1 }
⟨latexrelease⟩              {
⟨latexrelease⟩                \cs:w __hook_toplevel~#1 \exp_after:wN \cs_end:
⟨latexrelease⟩                \cs:w __hook_next~#1 \cs_end:
⟨latexrelease⟩              }
⟨latexrelease⟩          }
⟨latexrelease⟩          {
⟨latexrelease⟩            \__hook_if_reversed:nTF {#1}
⟨latexrelease⟩              { \cs_set_eq:NN \__hook_tl_gput:Nn    \__hook_tl_gput_left:Nn
⟨latexrelease⟩                \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_left:NV  }
⟨latexrelease⟩              { \cs_set_eq:NN \__hook_tl_gput:Nn    \__hook_tl_gput_right:Nn
⟨latexrelease⟩                \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_right:NV }
⟨latexrelease⟩            \prop_set_eq:Nc \l__hook_work_prop { g__hook_#1_code_prop }
⟨latexrelease⟩            \__hook_initialize_single:ccn
⟨latexrelease⟩              { __hook~#1 } { g__hook_#1_labels_clist } {#1}
⟨latexrelease⟩            \__hook_debug:n
⟨latexrelease⟩              { \exp_args:NNx \prop_gput:Nnn \g__hook_used_prop {#1} { } }
⟨latexrelease⟩          }
⟨latexrelease⟩      }
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
```

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

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

FMi: note to myself: verify

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

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

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

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

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

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

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

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

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

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

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

(*End of 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-cl*
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

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_initialize_single:NNn}
⟨latexrelease⟩                        {Hooks~with~args}
\cs_new_protected:Npn \__hook_initialize_single:NNn #1#2#3
  {
```

Step T1: Initialize the data structure …

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

Store the name of the hook:

```
    \tl_set:Nn \l__hook_cur_hook_tl {#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.

```
    \prop_map_inline:Nn \l__hook_work_prop
      {
        \int_incr:N \l__hook_labels_int
        \seq_put_right:Nn \l__hook_labels_seq {##1}
        \__hook_tl_set:cn { \__hook_tl_csname:n {##1} } { 0 }
        \seq_clear_new:c { \__hook_seq_csname:n {##1} }
      }
```

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.

```
    \prop_map_inline:Nn \l__hook_work_prop
     {
       \prop_map_inline:Nn \l__hook_work_prop
         {
           \__hook_if_label_case:nnnnn {##1} {####1}
             { \prop_map_break: }
             { \__hook_apply_label_pair:nnn {##1} {####1} }
             { \__hook_apply_label_pair:nnn {####1} {##1} }
                 {#3}
         }
     }
```

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

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

Step T4:

```
    \tl_set:Nn \l__hook_rear_tl { 0 }
    \tl_set:cn { \__hook_tl_csname:n { 0 } } { 0 }
    \seq_map_inline:Nn \l__hook_labels_seq
      {
        \int_compare:nNnT { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
            {
              \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } }{##1}
              \tl_set:Nn \l__hook_rear_tl {##1}
            }
      }
    \tl_set_eq:Nc \l__hook_front_tl { \__hook_tl_csname:n { 0 } }
```

```
    \__hook_tl_gclear:N #1
    \clist_gclear:N #2
```

The whole loop gets combined in steps T5–T7:

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

This part is step T5:

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

```
        \__hook_clist_gput:NV #2 \l__hook_front_tl
        \__hook_debug:n{ \iow_term:x{Handled~ code~ for~ \l__hook_front_tl} }
```

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

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

and here is step T7:

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

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

```
    }
  \int_compare:nNnF \l__hook_labels_int = 0
    {
      \iow_term:x{====================}
      \iow_term:x{Error:~ label~ rules~ are~ incompatible:}
```

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

FMi: improve output on a rainy day

```
      \__hook_debug_label_data:N \l__hook_work_prop
      \iow_term:x{====================}
    }
```

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 (to the right). The hook may take arguments, so we add a run of braced parameters after the `_next` and `_toplevel` macros, so that the arguments passed to the hook are forwarded to them.

```
  \exp_args:NNe \__hook_tl_gput:Nn #1
    { \exp_not:c { __hook_toplevel~#3 } \__hook_braced_parameter:n {#3} }
  \__hook_tl_gput_right:Ne #1
    { \exp_not:c { __hook_next~#3 } \__hook_braced_parameter:n {#3} }
  \use:e
    {
      \cs_gset:cpn { __hook~#3 } \use:c { c__hook_#3_parameter_tl }
        { \exp_not:V #1 }
    }
 }
\cs_generate_variant:Nn \__hook_initialize_single:NNn { cc }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_initialize_single:NNn}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_new_protected:Npn \__hook_initialize_single:NNn #1#2#3
⟨latexrelease⟩  {
⟨latexrelease⟩    \seq_clear:N \l__hook_labels_seq
⟨latexrelease⟩    \int_zero:N  \l__hook_labels_int
⟨latexrelease⟩    \tl_set:Nn \l__hook_cur_hook_tl {#3}
⟨latexrelease⟩    \prop_map_inline:Nn \l__hook_work_prop
⟨latexrelease⟩       {
⟨latexrelease⟩         \int_incr:N \l__hook_labels_int
⟨latexrelease⟩         \seq_put_right:Nn \l__hook_labels_seq {##1}
⟨latexrelease⟩         \__hook_tl_set:cn { \__hook_tl_csname:n {##1} } { 0 }
⟨latexrelease⟩         \seq_clear_new:c { \__hook_seq_csname:n {##1} }
⟨latexrelease⟩       }
⟨latexrelease⟩    \prop_map_inline:Nn \l__hook_work_prop
⟨latexrelease⟩      {
⟨latexrelease⟩        \prop_map_inline:Nn \l__hook_work_prop
⟨latexrelease⟩          {
⟨latexrelease⟩            \__hook_if_label_case:nnnnn {##1} {####1}
⟨latexrelease⟩              { \prop_map_break: }
⟨latexrelease⟩              { \__hook_apply_label_pair:nnn {##1} {####1} }
⟨latexrelease⟩              { \__hook_apply_label_pair:nnn {####1} {##1} }
⟨latexrelease⟩                  {#3}
⟨latexrelease⟩          }
⟨latexrelease⟩      }
⟨latexrelease⟩    \__hook_debug:n { \__hook_debug_label_data:N \l__hook_work_prop }
⟨latexrelease⟩    \tl_set:Nn \l__hook_rear_tl { 0 }
⟨latexrelease⟩    \tl_set:cn { \__hook_tl_csname:n { 0 } } { 0 }
⟨latexrelease⟩    \seq_map_inline:Nn \l__hook_labels_seq
⟨latexrelease⟩      {
⟨latexrelease⟩        \int_compare:nNnT { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
⟨latexrelease⟩            {
⟨latexrelease⟩              \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } }{##1}
⟨latexrelease⟩              \tl_set:Nn \l__hook_rear_tl {##1}
⟨latexrelease⟩            }
⟨latexrelease⟩      }
⟨latexrelease⟩    \tl_set_eq:Nc \l__hook_front_tl { \__hook_tl_csname:n { 0 } }
⟨latexrelease⟩    \__hook_tl_gclear:N #1
⟨latexrelease⟩    \clist_gclear:N #2
⟨latexrelease⟩    \bool_while_do:nn { ! \str_if_eq_p:Vn \l__hook_front_tl { 0 } }
⟨latexrelease⟩      {
⟨latexrelease⟩        \int_decr:N \l__hook_labels_int
⟨latexrelease⟩        \prop_get:NVN \l__hook_work_prop \l__hook_front_tl \l__hook_return_tl
⟨latexrelease⟩        \exp_args:NNV \__hook_tl_gput:Nn #1 \l__hook_return_tl
⟨latexrelease⟩        \__hook_clist_gput:NV #2 \l__hook_front_tl
⟨latexrelease⟩        \__hook_debug:n{ \iow_term:x{Handled~ code~ for~ \l__hook_front_tl} }
⟨latexrelease⟩        \seq_map_inline:cn { \__hook_seq_csname:n { \l__hook_front_tl } }
⟨latexrelease⟩          {
⟨latexrelease⟩            \tl_set:cx { \__hook_tl_csname:n {##1} }
⟨latexrelease⟩                       { \int_eval:n
⟨latexrelease⟩                           { \cs:w \__hook_tl_csname:n {##1} \cs_end: - 1 }
⟨latexrelease⟩                       }
⟨latexrelease⟩            \int_compare:nNnT
⟨latexrelease⟩                { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
⟨latexrelease⟩                {
⟨latexrelease⟩                  \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } } {##1}
⟨latexrelease⟩                  \tl_set:Nn \l__hook_rear_tl            {##1}
⟨latexrelease⟩                }
⟨latexrelease⟩          }
⟨latexrelease⟩        \tl_set_eq:Nc \l__hook_front_tl
⟨latexrelease⟩                      { \__hook_tl_csname:n { \l__hook_front_tl } }
⟨latexrelease⟩      }
⟨latexrelease⟩    \int_compare:nNnF \l__hook_labels_int = 0
⟨latexrelease⟩      {
⟨latexrelease⟩        \iow_term:x{====================}
⟨latexrelease⟩        \iow_term:x{Error:~ label~ rules~ are~ incompatible:}
⟨latexrelease⟩        \__hook_debug_label_data:N \l__hook_work_prop
⟨latexrelease⟩        \iow_term:x{====================}
⟨latexrelease⟩      }
⟨latexrelease⟩    \exp_args:NNo \__hook_tl_gput:Nn #1 { \cs:w __hook_toplevel~#3 \cs_end: }
⟨latexrelease⟩    \__hook_tl_gput_right:No #1 { \cs:w __hook_next~#3 \cs_end: }
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_generate_variant:Nn \__hook_tl_gput_right:Nn { No }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of 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.

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

(*End of 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`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(*End of 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.[9] So we still unnecessarily try to sort it in and depending on the rules that might result in a loop that is otherwise resolved. If that turns out to be a real issue, we can improve the code.

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

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

[9]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.

```
  }
```

(*End of 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_xW:nnn`
`\__hook_apply_-rule_xE:nnn`

Reversed rules.

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

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

`\__hook_msg_pair_found:nnn`

A macro to avoid moving this many tokens around.

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

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

`\__hook_debug_label_data:N`

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

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

`\hook_show:n`
`\hook_log:n`
`\__hook_log_line:x`
`\__hook_log_line_indent:x`
`\__hook_log:nN`

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

```
\cs_new_protected:Npn \hook_log:n #1
  {
    \cs_set_eq:NN \__hook_log_cmd:x \iow_log:x
    \__hook_normalize_hook_args:Nn \__hook_log:nN {#1} \tl_log:x
  }
\cs_new_protected:Npn \hook_show:n #1
  {
    \cs_set_eq:NN \__hook_log_cmd:x \iow_term:x
    \__hook_normalize_hook_args:Nn \__hook_log:nN {#1} \tl_show:x
  }
\cs_new_protected:Npn \__hook_log_line:x #1
  { \__hook_log_cmd:x { >~#1 } }
\cs_new_protected:Npn \__hook_log_line_indent:x #1
  { \__hook_log_cmd:x { >~\@spaces #1 } }
```

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_log:nN}
⟨latexrelease⟩                 {Hooks~with~args}
\cs_new_protected:Npn \__hook_log:nN #1 #2
  {
    \__hook_if_deprecated_generic:nT {#1}
      {
        \__hook_deprecated_generic_warn:n {#1}
        \__hook_do_deprecated_generic:Nn \__hook_log:nN {#1} #2
        \exp_after:wN \use_none:nnnnnnnnn \use_none:nnnnn
      }
    \__hook_preamble_hook:n {#1}
    \__hook_log_cmd:x
      {
        ^^J ->~The~
        \__hook_if_generic:nT {#1} { generic~ }
        hook~'#1'
        \__hook_if_disabled:nF {#1}
          {
            \exp_args:Nf \__hook_print_args:nn {#1}
              {
                \int_eval:n
                  { \str_count:e { \__hook_parameter:n {#1} } / 3 }
              }
          }
        :
      }
    \__hook_if_usable:nF {#1}
      { \__hook_log_line:x { The~hook~is~not~declared. } }
    \__hook_if_disabled:nT {#1}
      { \__hook_log_line:x { The~hook~is~disabled. } }
    \hook_if_empty:nTF {#1}
      { #2 { The~hook~is~empty } }
      {
        \__hook_log_line:x { Code~chunks: }
        \prop_if_empty:cTF { g__hook_#1_code_prop }
          { \__hook_log_line_indent:x { --- } }
          {
            \prop_map_inline:cn { g__hook_#1_code_prop }
              {
                \exp_after:wN \cs_set:Npn \exp_after:wN \__hook_tmp:w
                  \c__hook_nine_parameters_tl {##2}
                \__hook_log_line_indent:x
                  { ##1~-->~\cs_replacement_spec:N \__hook_tmp:w }
              }
          }
```

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

```
        \__hook_log_line:x
          {
            Document-level~(top-level)~code
            \__hook_if_usable:nT {#1}
              { ~(executed~\__hook_if_reversed:nTF {#1} {first} {last} ) } :
          }
        \__hook_log_line_indent:x
          {
            \__hook_cs_if_empty:cTF { __hook_toplevel~#1 }
              { --- }
              { -> ~ \cs_replacement_spec:c { __hook_toplevel~#1 } }
          }
```

```
        \__hook_log_line:x { Extra~code~for~next~invocation: }
        \__hook_log_line_indent:x
          {
            \__hook_cs_if_empty:cTF { __hook_next~#1 }
              { --- }
```

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.

```
              {
                -> ~ \exp_last_unbraced:Nf \__hook_log_next_code:w
                  { \cs_replacement_spec:c { __hook_next~#1 } }
```

```
            }
          }
```

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.

```
        \__hook_log_line:x { Rules: }
        \bool_set_true:N \l__hook_tmpa_bool
        \__hook_list_rules:nn {#1}
          {
            \bool_set_false:N \l__hook_tmpa_bool
            \__hook_log_line_indent:x
              {
                ##2~ with~
                \str_if_eq:nnT {##3} {??} { default~ }
                relation~ ##1
              }
          }
        \bool_if:NT \l__hook_tmpa_bool
          { \__hook_log_line_indent:x { --- } }
```

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

```
        \bool_lazy_and:nnTF
            { \__hook_if_usable_p:n {#1} }
            { ! \hook_if_empty_p:n {#1} }
          {
            \__hook_log_line:x
              {
                Execution~order
                \bool_if:NTF \l__hook_tmpa_bool
                  { \__hook_if_reversed:nT {#1} { ~(after~reversal) } }
                  { ~(after~
                    \__hook_if_reversed:nT {#1} { reversal~and~ }
                    applying~rules)
                  } :
              }
            #2 % \tl_show:n
              {
                \@spaces
                \clist_if_empty:cTF { g__hook_#1_labels_clist }
                  { --- }
                  { \clist_use:cn { g__hook_#1_labels_clist } { ,~ } }
            }
          }
          {
            \__hook_log_line:x { Execution~order: }
            #2
              {
                \@spaces Not~set~because~the~hook~ \__hook_if_usable:nTF {#1}
                  { code~pool~is~empty }
                  { is~\__hook_if_disabled:nTF {#1} {disabled} {undeclared} }
              }
          }
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
%
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_log:nN}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_new_protected:Npn \__hook_log:nN #1 #2
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_deprecated_generic:nT {#1}
⟨latexrelease⟩      {
⟨latexrelease⟩        \__hook_deprecated_generic_warn:n {#1}
⟨latexrelease⟩        \__hook_do_deprecated_generic:Nn \__hook_log:nN {#1} #2
⟨latexrelease⟩        \exp_after:wN \use_none:nnnnnnnnn \use_none:nnnnn
⟨latexrelease⟩      }
⟨latexrelease⟩    \__hook_preamble_hook:n {#1}
⟨latexrelease⟩    \__hook_log_cmd:x
⟨latexrelease⟩      { ^^J ->~The~ \__hook_if_generic:nT {#1} { generic~ } hook~'#1': }
⟨latexrelease⟩    \__hook_if_usable:nF {#1}
⟨latexrelease⟩      { \__hook_log_line:x { The~hook~is~not~declared. } }
⟨latexrelease⟩    \__hook_if_disabled:nT {#1}
⟨latexrelease⟩      { \__hook_log_line:x { The~hook~is~disabled. } }
⟨latexrelease⟩    \hook_if_empty:nTF {#1}
⟨latexrelease⟩      { #2 { The~hook~is~empty } }
⟨latexrelease⟩      {
⟨latexrelease⟩        \__hook_log_line:x { Code~chunks: }
⟨latexrelease⟩        \prop_if_empty:cTF { g__hook_#1_code_prop }
⟨latexrelease⟩          { \__hook_log_line_indent:x { --- } }
⟨latexrelease⟩          {
⟨latexrelease⟩            \prop_map_inline:cn { g__hook_#1_code_prop }
⟨latexrelease⟩              { \__hook_log_line_indent:x { ##1~-->~\tl_to_str:n {##2} } }
⟨latexrelease⟩          }
```

```
⟨latexrelease⟩        \__hook_log_line:x
⟨latexrelease⟩          {
⟨latexrelease⟩            Document-level~(top-level)~code
⟨latexrelease⟩            \__hook_if_usable:nT {#1}
⟨latexrelease⟩              { ~(executed~\__hook_if_reversed:nTF {#1} {first} {last} ) } :
⟨latexrelease⟩          }
⟨latexrelease⟩        \__hook_log_line_indent:x
⟨latexrelease⟩          {
⟨latexrelease⟩            \tl_if_empty:cTF { __hook_toplevel~#1 }
⟨latexrelease⟩              { --- }
⟨latexrelease⟩              { -> ~ \exp_args:Nv \tl_to_str:n { __hook_toplevel~#1 } }
⟨latexrelease⟩          }
⟨latexrelease⟩        \__hook_log_line:x { Extra~code~for~next~invocation: }
⟨latexrelease⟩        \__hook_log_line_indent:x
⟨latexrelease⟩          {
⟨latexrelease⟩            \tl_if_empty:cTF { __hook_next~#1 }
⟨latexrelease⟩              { --- }
⟨latexrelease⟩              { ->~ \exp_args:Nv \__hook_log_next_code:n { __hook_next~#1 } }
⟨latexrelease⟩          }
⟨latexrelease⟩        \__hook_log_line:x { Rules: }
⟨latexrelease⟩        \bool_set_true:N \l__hook_tmpa_bool
⟨latexrelease⟩        \__hook_list_rules:nn {#1}
⟨latexrelease⟩          {
⟨latexrelease⟩            \bool_set_false:N \l__hook_tmpa_bool
⟨latexrelease⟩            \__hook_log_line_indent:x
⟨latexrelease⟩              {
⟨latexrelease⟩                ##2~ with~
⟨latexrelease⟩                \str_if_eq:nnT {##3} {??} { default~ }
⟨latexrelease⟩                relation~ ##1
⟨latexrelease⟩              }
⟨latexrelease⟩          }
⟨latexrelease⟩        \bool_if:NT \l__hook_tmpa_bool
⟨latexrelease⟩          { \__hook_log_line_indent:x { --- } }
⟨latexrelease⟩        \bool_lazy_and:nnTF
⟨latexrelease⟩            { \__hook_if_usable_p:n {#1} }
⟨latexrelease⟩            { ! \hook_if_empty_p:n {#1} }
⟨latexrelease⟩          {
⟨latexrelease⟩            \__hook_log_line:x
⟨latexrelease⟩              {
⟨latexrelease⟩                Execution~order
⟨latexrelease⟩                \bool_if:NTF \l__hook_tmpa_bool
⟨latexrelease⟩                  { \__hook_if_reversed:nT {#1} { ~(after~reversal) } }
⟨latexrelease⟩                     { ~(after~
⟨latexrelease⟩                       \__hook_if_reversed:nT {#1} { reversal~and~ }
⟨latexrelease⟩                       applying~rules)
⟨latexrelease⟩                     } :
⟨latexrelease⟩                 }
⟨latexrelease⟩               #2 % \tl_show:n
⟨latexrelease⟩                 {
⟨latexrelease⟩                   \@spaces
⟨latexrelease⟩                   \clist_if_empty:cTF { g__hook_#1_labels_clist }
⟨latexrelease⟩                     { --- }
⟨latexrelease⟩                     { \clist_use:cn { g__hook_#1_labels_clist } { ,~ } }
⟨latexrelease⟩                 }
⟨latexrelease⟩             }
⟨latexrelease⟩             {
⟨latexrelease⟩               \__hook_log_line:x { Execution~order: }
⟨latexrelease⟩               #2
⟨latexrelease⟩                 {
⟨latexrelease⟩                   \@spaces Not~set~because~the~hook~ \__hook_if_usable:nTF {#1}
⟨latexrelease⟩                     { code~pool~is~empty }
⟨latexrelease⟩                     { is~\__hook_if_disabled:nTF {#1} {disabled} {undeclared} }
⟨latexrelease⟩                 }
⟨latexrelease⟩             }
⟨latexrelease⟩         }
⟨latexrelease⟩   }
⟨latexrelease⟩\EndIncludeInRelease
```

To display the code for next invocation only (i.e., from `\AddToHookNext` we have to remove the string `\__hook_clear_next:n{`⟨*hook*⟩`}`, so the simplest is to use a macro delimited by a `}`$_1$2.

`\__hook_log_next_code:n`

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_log_next_code:n}
⟨latexrelease⟩                 {Hooks~with~args}
\exp_last_unbraced:NNNNo
\cs_new:Npn \__hook_log_next_code:w #1 \c_right_brace_str { }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_log_next_code:n}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_gset:Npn \__hook_log_next_code:n #1
⟨latexrelease⟩  { \exp_args:No \tl_to_str:n { \use_none:nn #1 } }
⟨latexrelease⟩\EndIncludeInRelease
```

Pretty-prints the number of arguments of a hook.

```
\cs_new:Npn \__hook_print_args:nn #1 #2
```

`\__hook_print_args:n`

```
  {
    \int_compare:nNnT {#2} > { 0 }
      {
        \__hook_if_declared:nT {#1} { \use_none:nnn }
        \__hook_if_cmd_hook:nT {#1}
          { \use_i:nnn { ~ (unknown ~ } }
        \use:n { ~ (#2 ~ }
        argument \int_compare:nNnT {#2} > { 1 } { s } )
      }
  }
```

(*End of definition for* `\hook_show:n` *and others. These functions are documented on page 22.*)

`\__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` = ⟨*label*$_1$⟩`|`⟨*label*$_2$⟩, and `#3` = ⟨*hook*⟩ (the latter may be the argument `#1` to `\__hook_list_rules:nn`, or `??` if it is a default rule).

```
\cs_new_protected:Npn \__hook_list_rules:nn #1 #2
  {
    \cs_set_protected:Npn \__hook_tmp:w ##1 ##2 ##3 {#2}
    \prop_map_inline:cn { g__hook_#1_code_prop }
      {
        \prop_map_inline:cn { g__hook_#1_code_prop }
          {
            \__hook_if_label_case:nnnnn {##1} {####1}
              { \prop_map_break: }
              { \__hook_list_one_rule:nnn {##1} {####1} }
              { \__hook_list_one_rule:nnn {####1} {##1} }
                  {#1}
          }
      }
  }
```

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*⟩.

```
\cs_new_protected:Npn \__hook_list_one_rule:nnn #1#2#3
  {
    \__hook_list_if_rule_exists:nnnF {#1} {#2} {#3}
      { \__hook_list_if_rule_exists:nnnF {#1} {#2} { ?? } { } }
  }
```

```
\cs_new_protected:Npn \__hook_list_if_rule_exists:nnnF #1#2#3
  {
    \if_cs_exist:w g__hook_ #3 _rule_ #1 | #2 _tl \cs_end:
      \exp_args:Nv \__hook_tmp:w
        { g__hook_ #3 _rule_ #1 | #2 _tl } { #1 | #2 } {#3}
      \exp_after:wN \use_none:nn
    \fi:
    \use:n
  }
```

(*End of 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`.

```
\cs_new_protected:Npn \__hook_debug_print_rules:n #1
  {
    \iow_term:n { The~hook~#1~contains~the~rules: }
    \cs_set_protected:Npn \__hook_tmp:w ##1
      {
        \__hook_list_rules:nn {#1}
          {
            \iow_term:x
              {
                > ##1 {####2} ##1 => ##1 {####1}
                \str_if_eq:nnT {####3} {??} { ~(default) }
              }
          }
      }
    \exp_args:No \__hook_tmp:w { \use:nn { ~ } { ~ } }
  }
```

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

4.8 Specifying code for next invocation

`\hook_gput_next_code:nn`

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_gput_next_code:nn}
⟨latexrelease⟩                           {Hooks~with~args}
\cs_new_protected:Npn \hook_gput_next_code:nn #1 #2
  {
    \__hook_replacing_args_false:
    \__hook_normalize_hook_args:Nn \__hook_gput_next_code:nn {#1} {#2}
    \__hook_replacing_args_reset:
  }
\cs_new_protected:Npn \hook_gput_next_code_with_args:nn #1 #2
  {
    \__hook_replacing_args_true:
    \__hook_normalize_hook_args:Nn \__hook_gput_next_code:nn {#1} {#2}
    \__hook_replacing_args_reset:
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_gput_next_code:nn}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \hook_gput_next_code:nn #1
⟨latexrelease⟩  { \__hook_normalize_hook_args:Nn \__hook_gput_next_code:nn {#1} }
⟨latexrelease⟩\cs_gset_protected:Npn \hook_gput_next_code_with_args:nn #1 #2 { }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_gput_next_code:nn`. *This function is documented on page* 20.)

`\__hook_gput_next_code:nn`

```
\cs_new_protected:Npn \__hook_gput_next_code:nn #1 #2
  {
    \__hook_if_disabled:nTF {#1}
      { \msg_error:nnn { hooks } { hook-disabled } {#1} }
      {
        \__hook_if_structure_exist:nTF {#1}
          { \__hook_gput_next_do:nn }
          { \__hook_try_declaring_generic_next_hook:nn }
              {#1} {#2}
      }
  }
```

(*End of definition for* `\__hook_gput_next_code:nn`.)

`\__hook_gput_next_do:nn` Start by sanity-checking with `\__hook_chk_args_allowed:nn`. Then 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.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_gput_next_do:nn}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new_protected:Npn \__hook_gput_next_do:nn #1
  {
    \__hook_init_structure:n {#1}
    \__hook_chk_args_allowed:nn {#1} { AddToHookNext }
    \__hook_cs_if_empty:cT { __hook~#1 }
      { \__hook_update_hook_code:n {#1} }
    \__hook_cs_if_empty:cT { __hook_next~#1 }
      { \__hook_next_gset:nn {#1} { \__hook_clear_next:n {#1} } }
    \__hook_cs_gput_right:nnn { _next } {#1}
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_gput_next_do:nn}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_gput_next_do:nn #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \exp_args:Nc \__hook_gput_next_do:Nnn
⟨latexrelease⟩      { __hook_next~#1 } {#1}
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_gput_next_do:Nnn #1 #2
⟨latexrelease⟩  {
⟨latexrelease⟩    \tl_if_empty:cT { __hook~#2 }
⟨latexrelease⟩      { \__hook_update_hook_code:n {#2} }
⟨latexrelease⟩    \tl_if_empty:NT #1
⟨latexrelease⟩      { \__hook_tl_gset:Nn #1 { \__hook_clear_next:n {#2} } }
⟨latexrelease⟩    \__hook_tl_gput_right:Nn #1
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_gput_next_do:nn`.)

`\hook_gclear_next_code:n` Discard anything set up for next invocation of the hook.

```
\cs_new_protected:Npn \hook_gclear_next_code:n #1
  { \__hook_normalize_hook_args:Nn \__hook_clear_next:n {#1} }
```

(*End of definition for* `\hook_gclear_next_code:n`. *This function is documented on page 20.*)

`\__hook_clear_next:n`

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_clear_next:n}
⟨latexrelease⟩                 {Hooks~with~args}
\cs_new_protected:Npn \__hook_clear_next:n #1
  { \__hook_next_gset:nn {#1} { } }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_clear_next:n}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_clear_next:n #1
⟨latexrelease⟩  { \cs_gset_eq:cN { __hook_next~#1 } \c_empty_tl }
```

```
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_clear_next:n`.)

4.9 Using the hook

`\hook_use:n`
`\__hook_use_initialized:n`
`\__hook_preamble_hook:n`

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

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

Both versions do the same 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.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_use:n}
⟨latexrelease⟩                             {Hooks~with~args}
\cs_new_protected:Npn \hook_use:n #1
  {
    \__hook_preamble_hook:n {#1}
    \__hook_use_initialized:n {#1}
  }
\cs_new:Npn \__hook_use_initialized:n #1
  {
    \if_cs_exist:w __hook~#1 \cs_end:
      \cs:w __hook~#1 \use_i:nn
    \fi:
    \use_none:n
    \cs_end:
  }
\cs_new_protected:Npn \__hook_preamble_hook:n #1
  {
    \if_cs_exist:w __hook~#1 \cs_end:
      \__hook_initialize_hook_code:n {#1}
    \fi:
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\hook_use:n}
⟨latexrelease⟩                 {Standardise~generic~hook~names}
⟨latexrelease⟩\cs_new_protected:Npn \hook_use:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \tl_if_exist:cT { __hook~#1 }
⟨latexrelease⟩      {
⟨latexrelease⟩        \__hook_preamble_hook:n {#1}
⟨latexrelease⟩        \cs:w __hook~#1 \cs_end:
⟨latexrelease⟩      }
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_new:Npn \__hook_use_initialized:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \if_cs_exist:w __hook~#1 \cs_end:
⟨latexrelease⟩      \cs:w __hook~#1 \exp_after:wN \cs_end:
⟨latexrelease⟩    \fi:
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_new_protected:Npn \__hook_preamble_hook:n #1
⟨latexrelease⟩  { \__hook_initialize_hook_code:n {#1} }
⟨latexrelease⟩\cs_new:Npn \hook_use:nnw #1 { }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_use:n}
⟨latexrelease⟩                 {Standardise~generic~hook~names}
⟨latexrelease⟩\cs_new_protected:Npn \hook_use:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \tl_if_exist:cTF { __hook~#1 }
⟨latexrelease⟩      {
⟨latexrelease⟩        \__hook_preamble_hook:n {#1}
⟨latexrelease⟩        \cs:w __hook~#1 \cs_end:
⟨latexrelease⟩      }
⟨latexrelease⟩      { \__hook_use:wn #1 / \s__hook_mark {#1} }
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_new:Npn \__hook_use_initialized:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \if_cs_exist:w __hook~#1 \cs_end:
⟨latexrelease⟩    \else:
⟨latexrelease⟩      \__hook_use_undefined:w
⟨latexrelease⟩    \fi:
⟨latexrelease⟩    \cs:w __hook~#1 \__hook_use_end:
```

```
⟨latexrelease⟩   }
⟨latexrelease⟩\cs_new:Npn \__hook_use_undefined:w #1 #2 __hook~#3 \__hook_use_end:
⟨latexrelease⟩   {
⟨latexrelease⟩     #1 % fi
⟨latexrelease⟩     \__hook_use:wn #3 / \s__hook_mark {#3}
⟨latexrelease⟩   }
⟨latexrelease⟩\cs_new_protected:Npn \__hook_preamble_hook:n #1
⟨latexrelease⟩   { \__hook_initialize_hook_code:n {#1} }
⟨latexrelease⟩\cs_new_eq:NN \__hook_use_end: \cs_end:
⟨latexrelease⟩\cs_new:Npn \hook_use:nnw #1 { }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_use:n`, `\__hook_use_initialized:n`, *and* `\__hook_preamble_hook:n`. *This function is documented on page 19.*)

\hook_use:nnw
__hook_use_initialized:nnw

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_use:nnw}
⟨latexrelease⟩                    {Hooks~with~args}
\cs_new_protected:Npn \hook_use:nnw #1
  {
    \__hook_preamble_hook:n {#1}
    \__hook_use_initialized:nnw {#1}
  }
\cs_new:Npn \__hook_use_initialized:nnw #1 #2
  {
    \cs:w
      \if_cs_exist:w __hook~#1 \cs_end:
        __hook~#1
      \else:
        use_none: \prg_replicate:nn {#2} { n }
      \fi:
    \cs_end:
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_use:nnw}
⟨latexrelease⟩                    {Hooks~with~args}
⟨latexrelease⟩\cs_gset:Npn \hook_use:nnw #1 #2
⟨latexrelease⟩   { \use:c { use_none: \prg_replicate:nn {#2} { n } } }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_use:nnw` *and* `\__hook_use_initialized:nnw`. *This function is documented on page 19.*)

__hook_post_initialization_defs:

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_post_initialization_defs:}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new_protected:Npn \__hook_post_initialization_defs:
  {
    \cs_gset_eq:NN \hook_use:n \__hook_use_initialized:n
    \cs_gset_eq:NN \hook_use:nnw \__hook_use_initialized:nnw
    \cs_gset_eq:NN \__hook_preamble_hook:n \use_none:n
    \cs_gset_eq:NN \__hook_post_initialization_defs: \prg_do_nothing:
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_post_initialization_defs:}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_post_initialization_defs:
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_post_initialization_defs:`.)

`\__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.

```
⟨latexrelease⟩\IncludeInRelease{2021/11/15}{\__hook_use:wn}
⟨latexrelease⟩                  {Standardise~generic~hook~names}
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_use:wn}
⟨latexrelease⟩                  {Standardise~generic~hook~names}
⟨latexrelease⟩\cs_new:Npn \__hook_use:wn #1 / #2 \s__hook_mark #3
⟨latexrelease⟩  {
⟨latexrelease⟩    \str_if_eq:nnTF {#1} { file }
⟨latexrelease⟩      { \__hook_try_file_hook:n {#3} }
⟨latexrelease⟩      { } % Hook doesn't exist
⟨latexrelease⟩  }

⟨latexrelease⟩\cs_new_protected:Npn \__hook_try_file_hook:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_file_hook:wTF #1 / \s__hook_mark
⟨latexrelease⟩      {
⟨latexrelease⟩        \exp_args:Ne \__hook_if_usable_use:n
⟨latexrelease⟩          { \exp_args:Ne \__hook_file_hook_normalize:n {#1} }
⟨latexrelease⟩        }
⟨latexrelease⟩        { \__hook_if_usable_use:n {#1} } % file/ generic hook (e.g. file/before)
⟨latexrelease⟩    }
```

```
⟨latexrelease⟩\cs_new_protected:Npn \__hook_if_usable_use:n #1
⟨latexrelease⟩    {
⟨latexrelease⟩      \tl_if_exist:cT { __hook~#1 }
⟨latexrelease⟩        {
⟨latexrelease⟩          \__hook_preamble_hook:n {#1}
⟨latexrelease⟩          \cs:w __hook~#1 \cs_end:
⟨latexrelease⟩        }
⟨latexrelease⟩    }
⟨latexrelease⟩\EndIncludeInRelease
```

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

`\hook_use_once:n`
`\hook_use_once:nnw`

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.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_use_once:nnw}
⟨latexrelease⟩                        {Hooks~with~args}
\cs_new_protected:Npn \hook_use_once:n #1
  {
    \__hook_if_execute_immediately:nF {#1}
      { \__hook_normalize_hook_args:Nn \__hook_use_once:nn { \use:n {#1} } { 0 } }
  }
\cs_new_protected:Npn \hook_use_once:nnw #1 #2
  {
    \__hook_if_execute_immediately:nF {#1}
      { \__hook_normalize_hook_args:Nn \__hook_use_once:nn { \use:n {#1} } {#2} }
  }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_use_once:n` *and* `\hook_use_once:nnw`*. These functions are documented on page 19.*)

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_use_once:nnw}
⟨latexrelease⟩                        {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \hook_use_once:n #1
⟨latexrelease⟩    {
```

```
⟨latexrelease⟩      \__hook_if_execute_immediately:nF {#1}
⟨latexrelease⟩        { \__hook_normalize_hook_args:Nn \__hook_use_once:n { \use:n {#1} } }
⟨latexrelease⟩    }
⟨latexrelease⟩\cs_gset:Npn \hook_use_once:nnw #1 #2
⟨latexrelease⟩  { \use:c { use_none: \prg_replicate:nn {#2} { n } } }
⟨latexrelease⟩\EndIncludeInRelease
```

__hook_use_once:nn

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_use_once:nn}
⟨latexrelease⟩                 {Hooks~with~args}
\cs_new_protected:Npn \__hook_use_once:nn #1 #2
  {
    \__hook_preamble_hook:n {#1}
    \__hook_use_once_set:n {#1}
```

When a hook has arguments, the call to `\__hook_use_initialized:n`, should be the very last thing to happen, otherwise the arguments grabbed will be wrong. So, to clean up after the hook we need to cheat a bit and sneak the cleanup code at the end of the hook, along with the next execution code.

```
    \__hook_replacing_args_false:
    \__hook_cs_gput_right:nnn { _next } {#1} { \__hook_use_once_clear:n {#1} }
    \__hook_replacing_args_reset:
    \__hook_if_usable:nTF {#1}
      { \__hook_use_initialized:n {#1} }
      {
        \int_compare:nNnT {#2} > { 0 }
          { \use:c { use_none: \prg_replicate:nn {#2} { n } } }
      }
  }
⟨latexrelease⟩\EndIncludeInRelease
%
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_use_once:nn}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_gset_protected:Npn \__hook_use_once:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_preamble_hook:n {#1}
⟨latexrelease⟩    \__hook_use_once_set:n {#1}
⟨latexrelease⟩    \__hook_use_initialized:n {#1}
⟨latexrelease⟩    \__hook_use_once_clear:n {#1}
⟨latexrelease⟩  }
⟨latexrelease⟩\cs_undefine:N \__hook_use_once:nn
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_use_once:nn`.)

`\__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.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_use_once_clear:n}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new_protected:Npn \__hook_use_once_set:n #1
  { \__hook_tl_gset:cn { g__hook_#1_reversed_tl } { I } }
\cs_new_protected:Npn \__hook_use_once_clear:n #1
  {
    \__hook_code_gset:nn {#1} { }
    \__hook_next_gset:nn {#1} { }
    \__hook_toplevel_gset:nn {#1} { }
    \prop_gclear_new:c { g__hook_#1_code_prop }
  }
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_use_once_clear:n}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_new_protected:Npn \__hook_use_once_clear:n #1
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_tl_gclear:c { __hook~#1 }
⟨latexrelease⟩    \__hook_tl_gclear:c { __hook_next~#1 }
⟨latexrelease⟩    \__hook_tl_gclear:c { __hook_toplevel~#1 }
⟨latexrelease⟩    \prop_gclear_new:c { g__hook_#1_code_prop }
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_use_once_set:n` *and* `\__hook_use_once_clear:n`.)

`\__hook_if_execute_immediately_p:n`
`\__hook_if_execute_immediately:nTF`

To check whether the code being added should be executed immediately (that is, if the hook is a one-time hook), we check if `\g__hook_`⟨*hook*⟩`_reversed_tl` is `I`. The gymnastics around `\if:w` is there to allow the `reversed` token list to be empty.

```
\prg_new_conditional:Npnn \__hook_if_execute_immediately:n #1 { T, F, TF }
  {
    \exp_after:wN \__hook_use_none_delimit_by_s_mark:w
    \if:w I
        \if_cs_exist:w g__hook_#1_reversed_tl \cs_end:
          \cs:w g__hook_#1_reversed_tl \exp_after:wN \cs_end:
        \fi:
        X
      \s__hook_mark \prg_return_true:
    \else:
      \s__hook_mark \prg_return_false:
    \fi:
  }
```

(*End of 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:nTF`

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.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\hook_if_empty:n}
⟨latexrelease⟩                         {Hooks~with~args}
\prg_new_conditional:Npnn \hook_if_empty:n #1 { p , T , F , TF }
  {
    \if:w
        T
        \prop_if_exist:cT { g__hook_#1_code_prop }
          { \prop_if_empty:cF { g__hook_#1_code_prop } { F } }
        \__hook_cs_if_empty:cF { __hook_toplevel~#1 } { F }
        \__hook_cs_if_empty:cF { __hook_next~#1 } { F }
        T
```

```
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\hook_if_empty:n}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\prg_new_conditional:Npnn \hook_if_empty:n #1 { p , T , F , TF }
⟨latexrelease⟩  {
⟨latexrelease⟩    \__hook_if_structure_exist:nTF {#1}
⟨latexrelease⟩      {
⟨latexrelease⟩        \bool_lazy_and:nnTF
⟨latexrelease⟩            { \prop_if_empty_p:c { g__hook_#1_code_prop } }
⟨latexrelease⟩            {
⟨latexrelease⟩              \bool_lazy_and_p:nn
⟨latexrelease⟩                { \tl_if_empty_p:c { __hook_toplevel~#1 } }
⟨latexrelease⟩                { \tl_if_empty_p:c { __hook_next~#1 } }
⟨latexrelease⟩            }
⟨latexrelease⟩          { \prg_return_true: }
⟨latexrelease⟩          { \prg_return_false: }
⟨latexrelease⟩      }
⟨latexrelease⟩      { \prg_return_true: }
⟨latexrelease⟩  }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\hook_if_empty:nTF`. *This function is documented on page 21.*)

`\__hook_if_usable_p:n`
`\__hook_if_usable:nTF`

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).

```
\prg_new_conditional:Npnn \__hook_if_usable:n #1 { p , T , F , TF }
  {
    \cs_if_exist:cTF { __hook~#1 }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End of definition for* `\__hook_if_usable:nTF`.)

__hook_if_structure_exist_p:n
__hook_if_structure_exist:nTF

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`.

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

(*End of 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.

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

(*End of 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.

```
\prg_new_conditional:Npnn \__hook_if_reversed:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__hook_use_none_delimit_by_s_mark:w
    \if:w - \cs:w g__hook_#1_reversed_tl \cs_end:
      \s__hook_mark \prg_return_true:
    \else:
      \s__hook_mark \prg_return_false:
    \fi:
  }
```

(*End of 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.

```
\prg_new_conditional:Npnn \__hook_if_generic:n #1 { T, TF }
  { \__hook_if_generic:w #1 / / / \s__hook_mark }
\cs_new:Npn \__hook_if_generic:w #1 / #2 / #3 / #4 \s__hook_mark
  {
    \cs_if_exist:cTF { c__hook_generic_#1/./#3_tl }
      { \prg_return_true: }
      { \prg_return_false: }
  }
\prg_new_conditional:Npnn \__hook_if_deprecated_generic:n #1 { T, TF }
  { \__hook_if_deprecated_generic:w #1 / / / \s__hook_mark }
\cs_new:Npn \__hook_if_deprecated_generic:w #1 / #2 / #3 / #4 \s__hook_mark
  {
    \cs_if_exist:cTF { c__hook_deprecated_#1/./#2_tl }
      {
        \tl_if_empty:nTF {#3}
          { \prg_return_false: }
          { \prg_return_true: }
      }
      { \prg_return_false: }
  }
```

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

`\__hook_if_cmd_hook_p:n`
`\__hook_if_cmd_hook:nTF`
`\__hook_if_cmd_hook_p:w`
`\__hook_if_cmd_hook:wTF`

An internal conditional that checks if a given hook is a valid generic `cmd` hook.

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\__hook_if_cmd_hook:n}
⟨latexrelease⟩                    {Hooks~with~args}
\prg_new_conditional:Npnn \__hook_if_cmd_hook:n #1 { T }
  { \__hook_if_cmd_hook:w #1 / / / \s__hook_mark }
\cs_new:Npn \__hook_if_cmd_hook:w #1 / #2 / #3 / #4 \s__hook_mark
  {
    \if:w Y
          \str_if_eq:nnF {#1} { cmd } { N }
          \tl_if_exist:cF { c__hook_generic_#1/./#3_tl } { N }
          Y
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\__hook_if_cmd_hook:n}
⟨latexrelease⟩                    {Hooks~with~args}
⟨latexrelease⟩\cs_undefine:N \__hook_if_cmd_hook:nT
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\__hook_if_cmd_hook:nTF` *and* `\__hook_if_cmd_hook:wTF`.)

`\__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.

```
\prg_new_conditional:Npnn \__hook_if_generic_reversed:n #1 { T }
  { \__hook_if_generic_reversed:w #1 / / / \scan_stop: }
\cs_new:Npn \__hook_if_generic_reversed:w #1 / #2 / #3 / #4 \scan_stop:
  {
    \if_charcode:w - \cs:w c__hook_generic_#1/./#3_tl \cs_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\__hook_if_generic_reversed:nTF`.)

`\__hook_if_replacing_args:TF`
`\__hook_misused_if_replacing_args:nn`
`\__hook_replacing_args_true:`
`\__hook_replacing_args_false:`
`\__hook_replacing_args_reset:`
`\g__hook_replacing_stack_seq`

An internal conditional that checks if the code being added to the hook contains arguments.

```
\seq_new:N \g__hook_replacing_stack_seq
\cs_new:Npn \__hook_misused_if_replacing_args:nn #1 #2
  {
    \msg_expandable_error:nnn { latex2e } { should-not-happen }
      { Misused~\__hook_if_replacing_args:. }
  }
\cs_new:Npn \__hook_if_replacing_args:TF
  { \__hook_misused_if_replacing_args:nn }
\cs_new_protected:Npn \__hook_replacing_args_true:
  {
    \seq_gpush:No \g__hook_replacing_stack_seq
      { \__hook_if_replacing_args:TF }
    \cs_set:Npn \__hook_if_replacing_args:TF { \use_i:nn }
  }
\cs_new_protected:Npn \__hook_replacing_args_false:
  {
    \seq_gpush:No \g__hook_replacing_stack_seq
      { \__hook_if_replacing_args:TF }
    \cs_set:Npn \__hook_if_replacing_args:TF { \use_ii:nn }
  }
\cs_new_protected:Npn \__hook_replacing_args_reset:
  {
    \seq_gpop:NN \g__hook_replacing_stack_seq \l__hook_return_tl
    \cs_gset_eq:NN \__hook_if_replacing_args:TF \l__hook_return_tl
  }
```

(*End of definition for* `\__hook_if_replacing_args:TF` *and others.*)

4.11 Messages

Hook errors are LaTeX kernel errors:

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

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

```
\prop_gput:Nnn \g_msg_module_type_prop { latex2e } { LaTeX }
\prop_gput:Nnn \g_msg_module_name_prop { latex2e } { kernel }
```

```
\msg_new:nnnn { hooks } { labels-incompatible }
  {
    Labels~'#1'~and~'#2'~are~incompatible
    \str_if_eq:nnF {#3} {??} { ~in~hook~'#3' } .~
    \int_compare:nNnTF {#4} = { 1 }
      { The~ code~ for~ both~ labels~ will~ be~ dropped. }
      { You~ may~ see~ errors~ later. }
  }
  { LaTeX~found~two~incompatible~labels~in~the~same~hook.~
    This~indicates~an~incompatibility~between~packages.  }
```

```
\msg_new:nnnn { hooks } { exists }
    { Hook~'#1'~ has~ already~ been~ declared. }
    { There~ already~ exists~ a~ hook~ declaration~ with~ this~
      name.\\
      Please~ use~ a~ different~ name~ for~ your~ hook.}
```

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{too-many-args}
⟨latexrelease⟩                 {Hooks~with~args}
```

```
\msg_new:nnnn { hooks } { too-many-args }
  { Too~many~arguments~for~hook~'#1'. }
  {
    You~tried~to~declare~a~hook~with~#2~arguments,~but~a~
    hook~can~only~have~up~to~nine.~LaTeX~will~define~this~
    hook~with~nine~arguments.
  }
```

```
\msg_new:nnnn { hooks } { without-args }
  { Hook~'#1'~has~no~arguments. }
  {
    You~tried~to~use~\iow_char:N\\#2WithArguments~
    on~a~hook~that~takes~no~arguments.\\
    Check~the~usage~of~the~hook~or~use~\iow_char:N\\#2~instead.\\
    \\
    LaTeX~will~use~\iow_char:N\\#2.
  }
```

```
\msg_new:nnnn { hooks } { one-time-args }
  { You~can't~have~arguments~in~used~one-time~hook~'#1'. }
  {
    You~tried~to~use~\iow_char:N\\#2WithArguments~
    on~a~one-time~hook~that~has~already~been~used.~
    You~have~to~add~the~code~before~the~hook~is~used,~
    or~add~the~code~without~arguments~using~\iow_char:N\\#2~instead.\\
    \\
    LaTeX~will~use~\iow_char:N\\#2.
  }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{too-many-args}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\EndIncludeInRelease
\msg_new:nnnn { hooks } { hook-disabled }
  { Cannot~add~code~to~disabled~hook~'#1'. }
  {
    The~hook~'#1'~you~tried~to~add~code~to~was~previously~disabled~
    with~\iow_char:N\\hook_disable_generic:n~or~\iow_char:N\\DisableGenericHook,~so~
    it~cannot~have~code~added~to~it.
  }
\msg_new:nnn { hooks } { empty-label }
  {
    Empty~code~label~\msg_line_context:.~
    Using~'\__hook_currname_or_default:'~instead.
  }
\msg_new:nnn { hooks } { no-default-label }
  {
    Missing~(empty)~default~label~\msg_line_context:. \\
    This~command~was~ignored.
  }
\msg_new:nnnn { hooks } { unknown-rule }
  {
    Unknown~ relationship~ '#3'~
    between~ labels~ '#2'~ and~ '#4'~
    \str_if_eq:nnF {#1} {??} { ~in~hook~'#1' }. ~
    Perhaps~ a~ misspelling?
  }
  {
    The~ relation~ used~ not~ known~ to~ the~ system.~ Allowed~ values~ are~
    'before'~ or~ '<',~
    'after'~ or~ '>',~
    'incompatible-warning',~
    'incompatible-error',~
    'voids'~ or~
    'unrelated'.
  }
\msg_new:nnnn { hooks } { rule-too-late }
  {
    Sorting~rule~for~'#1'~hook~applied~too~late.\\
    Try~setting~this~rule~earlier.
  }
  {
    You~tried~to~set~the~ordering~of~hook~'#1'~using\\
    \ \ \iow_char:N\\DeclareHookRule{#1}{#2}{#3}{#4}\\
    but~hook~'#1'~was~already~used~as~a~one-time~hook,~
    thus~sorting~is\\
    no~longer~possible.~Declare~the~rule~
    before~the~hook~is~used.
  }
\msg_new:nnnn { hooks } { misused-top-level }
  {
    Illegal~use~of~\iow_char:N \\AddToHook{#1}[top-level]{...}.\\
    'top-level'~is~reserved~for~the~user's~document.
  }
  {
    The~'top-level'~label~is~meant~for~user~code~only,~and~should~only~
    be~used~(sparingly)~in~the~main~document.~Use~the~default~label~
    '\__hook_currname_or_default:'~for~this~\@cls@pkg,~or~another~
    suitable~label.
  }
\msg_new:nnn { hooks } { set-top-level }
  {
    You~cannot~change~the~default~label~#1~'top-level'.~Illegal \\
    \use:nn { ~ } { ~ } \iow_char:N \\#2{#3} \\
    \msg_line_context:.
  }
\msg_new:nnn { hooks } { extra-pop-label }
  {
    Extra~\iow_char:N \\PopDefaultHookLabel. \\
    This~command~will~be~ignored.
  }
```

```
\msg_new:nnn { hooks } { missing-pop-label }
  {
    Missing~\iow_char:N \\PopDefaultHookLabel. \\
    The~label~'#1'~was~pushed~but~never~popped.~Something~is~wrong.
  }
\msg_new:nnn { latex2e } { should-not-happen }
  {
    This~should~not~happen.~#1 \\
    Please~report~at~https://github.com/latex3/latex2e.
  }
\msg_new:nnn { hooks } { activate-disabled }
  {
    Cannot~ activate~ hook~ '#1'~ because~ it~ is~ disabled!
  }
\msg_new:nnn { hooks } { cannot-remove }
  {
    Cannot~remove~chunk~'#2'~from~hook~'#1'~because~
    \__hook_if_structure_exist:nTF {#1}
      { it~does~not~exist~in~that~hook. }
      { the~hook~does~not~exist. }
  }
\msg_new:nnn { hooks } { generic-deprecated }
  {
    Generic~hook~'#1/#2/#3'~is~deprecated. \\
    Use~hook~'#1/#3/#2'~instead.
  }
```

4.12 LaTeX 2ε package interface commands

\NewHook
\NewReversedHook
\NewMirroredHookPair

Declaring new hooks ...

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

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

\NewHookWithArguments
\NewReversedHookWithArguments
\NewMirroredHookPairWithArguments

Declaring new hooks with arguments...

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\NewHookWithArguments}
⟨latexrelease⟩                    {Hooks~with~args}
\NewDocumentCommand \NewHookWithArguments              { mm }
  { \hook_new_with_args:nn {#1} {#2} }
\NewDocumentCommand \NewReversedHookWithArguments     { mm }
  { \hook_new_reversed_with_args:nn {#1} {#2} }
\NewDocumentCommand \NewMirroredHookPairWithArguments { mmm }
  { \hook_new_pair_with_args:nnn {#1} {#2} {#3} }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\NewHookWithArguments}
⟨latexrelease⟩                    {Hooks~with~args}
⟨latexrelease⟩\cs_new_protected:Npn \NewHookWithArguments #1 #2 { }
⟨latexrelease⟩\cs_new_protected:Npn \NewReversedHookWithArguments #1 #2 { }
⟨latexrelease⟩\cs_new_protected:Npn \NewMirroredHookPairWithArguments #1 #2 #3 { }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\NewHookWithArguments`, `\NewReversedHookWithArguments`, *and* `\NewMirroredHookPairWithArguments`. *These functions are documented on page 4.*)

```
⟨latexrelease⟩\IncludeInRelease{2021/06/01}{\ActivateGenericHook}
⟨latexrelease⟩                    {Providing~hooks}
```

`\ActivateGenericHook` Providing new hooks …

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

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

`\DisableGenericHook` Disabling a generic hook.

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

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

```
⟨latexrelease⟩\EndIncludeInRelease
```

```
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\ActivateGenericHook}
⟨latexrelease⟩                    {Providing~hooks}
⟨latexrelease⟩\def \ActivateGenericHook #1 { }
⟨latexrelease⟩\def \DisableGenericHook #1 { }
⟨latexrelease⟩\EndIncludeInRelease
```

`\AddToHook`
`\AddToHookWithArguments`

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\AddToHookWithArguments}
⟨latexrelease⟩                    {Hooks~with~args}
\NewDocumentCommand \AddToHook { m o +m }
```

```
  { \hook_gput_code:nnn {#1} {#2} {#3} }
\NewDocumentCommand \AddToHookWithArguments { m o +m }
  { \hook_gput_code_with_args:nnn {#1} {#2} {#3} }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\AddToHookWithArguments}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_new_protected:Npn \AddToHookWithArguments #1 #2 #3 { }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\AddToHook` *and* `\AddToHookWithArguments`*. These functions are documented on page 7.*)

\AddToHookNext
\AddToHookNextWithArguments

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\AddToHookNextWithArguments}
⟨latexrelease⟩                 {Hooks~with~args}
\NewDocumentCommand \AddToHookNext { m +m }
  { \hook_gput_next_code:nn {#1} {#2} }
\NewDocumentCommand \AddToHookNextWithArguments { m +m }
  { \hook_gput_next_code_with_args:nn {#1} {#2} }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\AddToHookNextWithArguments}
⟨latexrelease⟩                 {Hooks~with~args}
⟨latexrelease⟩\cs_new_protected:Npn \AddToHookNextWithArguments #1 #2 { }
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\AddToHookNext` *and* `\AddToHookNextWithArguments`*. These functions are documented on page 9.*)

\ClearHookNext

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

(*End of definition for* `\ClearHookNext`*. This function is documented on page 10.*)

\RemoveFromHook

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

(*End of definition for* `\RemoveFromHook`*. This function is documented on page 8.*)

\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.

```
\NewDocumentCommand \SetDefaultHookLabel { m }
  { \__hook_set_default_hook_label:n {#1} }
%
```

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

```
\cs_gset_protected:Npn \@expl@push@filename@aux@@ #1#2#3
  {
    \__hook_curr_name_push:n {#3}
    \str_gset:Nx \g_file_curr_name_str {#3}
    #1 #2 {#3}
  }
```

(*End of definition for* `\SetDefaultHookLabel`*,* `\PushDefaultHookLabel`*, and* `\PopDefaultHookLabel`*. These functions are documented on page 13.*)

`\UseHook` `\UseOneTimeHook` `\UseHookWithArguments` `\UseOneTimeHookWithArguments`

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

```
⟨latexrelease⟩\IncludeInRelease{2023/06/01}{\UseHookWithArguments}
⟨latexrelease⟩                  {Hooks~with~args}
\cs_new:Npn \UseHook        { \hook_use:n }
\cs_new:Npn \UseOneTimeHook { \hook_use_once:n }
\cs_new:Npn \UseHookWithArguments        { \hook_use:nnw }
\cs_new:Npn \UseOneTimeHookWithArguments { \hook_use_once:nnw }
⟨latexrelease⟩\EndIncludeInRelease
⟨latexrelease⟩\IncludeInRelease{2020/10/01}{\UseHookWithArguments}
⟨latexrelease⟩                  {Hooks~with~args}
⟨latexrelease⟩\cs_new:Npn \UseHookWithArguments #1 #2 { }
⟨latexrelease⟩\cs_new:Npn \UseOneTimeHookWithArguments #1 #2 { }
```

```
⟨latexrelease⟩\EndIncludeInRelease
```

(*End of definition for* `\UseHook` *and others. These functions are documented on page 5.*)

`\ShowHook`
`\LogHook`

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

(*End of definition for* `\ShowHook` *and* `\LogHook`*. These functions are documented on page 16.*)

`\DebugHooksOn`
`\DebugHooksOff`

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

(*End of definition for* `\DebugHooksOn` *and* `\DebugHooksOff`*. These functions are documented on page 18.*)

`\DeclareHookRule`

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

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

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

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

(*End of definition for* `\DeclareDefaultHookRule`*. This function is documented on page 15.*)

`\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?

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

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

`\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.

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

(*End of definition for* `\IfHookEmptyTF`*. This function is documented on page 16.*)

`\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.*

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

(*End of definition for* `\IfHookExistsTF`*.*)

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.

```
\cs_new_protected:Npn \hook_disable:n
  {
    \__hook_deprecated_warn:nn
      { hook_disable:n }
      { hook_disable_generic:n }
    \hook_disable_generic:n
  }
\cs_new_protected:Npn \hook_provide:n
  {
    \__hook_deprecated_warn:nn
      { hook_provide:n }
      { hook_activate_generic:n }
    \hook_activate_generic:n
  }
\cs_new_protected:Npn \hook_provide_reversed:n
  {
    \__hook_deprecated_warn:nn
      { hook_provide_reversed:n }
      { hook_activate_generic:n }
    \__hook_activate_generic_reversed:n
  }
\cs_new_protected:Npn \hook_provide_pair:nn
  {
    \__hook_deprecated_warn:nn
      { hook_provide_pair:nn }
      { hook_activate_generic:n }
    \__hook_activate_generic_pair:nn
  }
\cs_new_protected:Npn \__hook_activate_generic_reversed:n #1
  { \__hook_normalize_hook_args:Nn \__hook_activate_generic:nn {#1} { - } }
\cs_new_protected:Npn \__hook_activate_generic_pair:nn #1#2
  { \hook_activate_generic:n {#1} \__hook_activate_generic_reversed:n {#2} }
```

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

\DisableHook \ProvideHook \ProvideReversedHook \ProvideMirroredHookPair

Deprecated.

```
\cs_new_protected:Npn \DisableHook
  {
    \__hook_deprecated_warn:nn
      { DisableHook }
      { DisableGenericHook }
    \hook_disable_generic:n
  }
\cs_new_protected:Npn \ProvideHook
  {
    \__hook_deprecated_warn:nn
      { ProvideHook }
      { ActivateGenericHook }
    \hook_activate_generic:n
  }
\cs_new_protected:Npn \ProvideReversedHook
  {
    \__hook_deprecated_warn:nn
      { ProvideReversedHook }
      { ActivateGenericHook }
    \__hook_activate_generic_reversed:n
  }
\cs_new_protected:Npn \ProvideMirroredHookPair
  {
    \__hook_deprecated_warn:nn
      { ProvideMirroredHookPair }
      { ActivateGenericHook }
    \__hook_activate_generic_pair:nn
  }
```

(*End of definition for* `\DisableHook` *and others.*)

__hook_deprecated_warn:nn

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

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

(*End of definition for* `\__hook_deprecated_warn:nn`.)

4.14 Internal commands needed elsewhere

Here we set up a few horrible (but consistent) LaTeX 2ε 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.

```
⟨@@=⟩
```

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

```
\cs_new_eq:NN \@expl@@@initialize@all@@
              \__hook_initialize_all:
\cs_new_eq:NN \@expl@@@hook@curr@name@pop@@
              \__hook_curr_name_pop:
```

(*End of definition for* `\@expl@@@initialize@all@@` *and* `\@expl@@@hook@curr@name@pop@@`.)

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.

```
%
⟨latexrelease⟩\IncludeInRelease{0000/00/00}{lthooks}
⟨latexrelease⟩                 {The~hook~management}%
⟨latexrelease⟩
⟨latexrelease⟩\def \NewHook#1{}
⟨latexrelease⟩\def \NewReversedHook#1{}
⟨latexrelease⟩\def \NewMirroredHookPair#1#2{}
⟨latexrelease⟩
⟨latexrelease⟩\def \DisableGenericHook #1{}
⟨latexrelease⟩
⟨latexrelease⟩\long\def\AddToHookNext#1#2{}
⟨latexrelease⟩
⟨latexrelease⟩\def\AddToHook#1{\@gobble@AddToHook@args}
⟨latexrelease⟩\providecommand\@gobble@AddToHook@args[2][]{}
⟨latexrelease⟩
⟨latexrelease⟩\def\RemoveFromHook#1{\@gobble@RemoveFromHook@arg}
⟨latexrelease⟩\providecommand\@gobble@RemoveFromHook@arg[1][]{}
⟨latexrelease⟩
⟨latexrelease⟩\def \UseHook        #1{}
⟨latexrelease⟩\def \UseOneTimeHook #1{}
⟨latexrelease⟩\def \ShowHook #1{}
⟨latexrelease⟩\let \DebugHooksOn \@empty
```

```
⟨latexrelease⟩\let \DebugHooksOff\@empty
⟨latexrelease⟩
⟨latexrelease⟩\def \DeclareHookRule #1#2#3#4{}
⟨latexrelease⟩\def \DeclareDefaultHookRule #1#2#3{}
⟨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.

```
⟨latexrelease⟩\long\def \IfHookExistsTF #1#2#3{#3}
⟨latexrelease⟩\long\def \IfHookEmptyTF #1#2#3{#2}
⟨latexrelease⟩
⟨latexrelease⟩\EndModuleRelease
```

```
⟨@@=hook⟩
```

```
⟨latexrelease⟩\cs:w __hook_rollback_tidying: \cs_end:
⟨latexrelease⟩\bool_lazy_and:nnT
⟨latexrelease⟩    { \int_compare_p:nNn { \sourceLaTeXdate } > { 20230600 } }
⟨latexrelease⟩    { \int_compare_p:nNn { \requestedLaTeXdate } < { 20230601 } }
⟨latexrelease⟩  {
⟨latexrelease⟩    \cs_gset_protected:Npn \__hook_rollback_tidying:
⟨latexrelease⟩      {
⟨latexrelease⟩        \@latex@error { Rollback~code~executed~twice }
⟨latexrelease⟩          {
⟨latexrelease⟩            Something~went~wrong~(unless~this~was~
⟨latexrelease⟩            done~on~purpose~in~a~testing~environment).
⟨latexrelease⟩          }
⟨latexrelease⟩        \use_none:nnnn
⟨latexrelease⟩      }
⟨latexrelease⟩    \cs_set:Npn \__hook_tmp:w #1 #2
⟨latexrelease⟩      {
⟨latexrelease⟩        \__hook_tl_gset:cx { __hook#1~#2 }
⟨latexrelease⟩          {
⟨latexrelease⟩            \exp_args:No \exp_not:o
⟨latexrelease⟩              {
⟨latexrelease⟩                \cs:w __hook#1~#2 \exp_last_unbraced:Ne \cs_end:
⟨latexrelease⟩                  { \__hook_braced_cs_parameter:n { __hook#1~#2 } }
⟨latexrelease⟩              }
⟨latexrelease⟩          }
⟨latexrelease⟩      }
⟨latexrelease⟩    \seq_map_inline:Nn \g__hook_all_seq
⟨latexrelease⟩      {
⟨latexrelease⟩        \exp_after:wN \cs_gset_nopar:Npn
⟨latexrelease⟩            \cs:w g__hook_#1_code_prop \exp_args:NNo \exp_args:No
⟨latexrelease⟩              \cs_end: { \cs:w g__hook_#1_code_prop \cs_end: }
⟨latexrelease⟩          \__hook_tmp:w { _toplevel } {#1}
⟨latexrelease⟩          \__hook_tmp:w { _next } {#1}
⟨latexrelease⟩        }
⟨latexrelease⟩    }
\ExplSyntaxOff
⟨/2ekernel | latexrelease⟩
⟨@@=⟩
```