

L^AT_EX’s hook management*

Frank Mittelbach[†]

October 30, 2026

Contents

1	Introduction	2
2	Package writer interface	2
2.1	L ^A T _E X 2 _ε interfaces	3
2.1.1	Declaring hooks	3
2.1.2	Special declarations for generic hooks	4
2.1.3	Using hooks in code	4
2.1.4	Updating code for hooks	5
2.1.5	Hook names and default labels	8
2.1.6	The <code>top-level</code> label	10
2.1.7	Defining relations between hook code	11
2.1.8	Querying hooks	13
2.1.9	Displaying hook code	14
2.1.10	Debugging hook code	15
2.2	L3 programming layer (<code>expl3</code>) interfaces	15
2.3	On the order of hook code execution	18
2.4	The use of “reversed” hooks	20
2.5	Difference between “normal” and “one-time” hooks	21
2.6	Generic hooks provided by packages	21
2.7	Hooks with arguments	22
2.8	Private L ^A T _E X kernel hooks	24
2.9	Legacy L ^A T _E X 2 _ε interfaces	24
3	L^AT_EX 2_ε commands and environments augmented by hooks	25
3.1	Generic hooks	25
3.1.1	Generic hooks for all environments	26
3.1.2	Generic hooks for commands	27
3.1.3	Generic hooks provided by file loading operations	27
3.2	Hooks provided by <code>\begin{document}</code>	28
3.3	Hooks provided by <code>\end{document}</code>	28
3.4	Hooks provided by <code>\shipout</code> operations	30
3.5	Hooks provided for paragraphs	30
3.6	Hooks provided in NFSS commands	30
3.7	Hook provided by the mark mechanism	31

*This module has version v1.1p dated 2026-06-01, © L^AT_EX Project.

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

4	The Implementation	31
4.1	Debugging	31
4.2	Borrowing from internals of other kernel modules	32
4.3	Declarations	32
4.4	Providing new hooks	34
4.4.1	The data structures of a hook	34
4.4.2	On the existence of hooks	35
4.4.3	Setting hooks up	36
4.4.4	Disabling and providing hooks	42
4.5	Parsing a label	44
4.6	Adding or removing hook code	48
4.7	Setting rules for hooks code	69
4.8	Specifying code for next invocation	91
4.9	Using the hook	93
4.10	Querying a hook	99
4.11	Messages	103
4.12	L ^A T _E X 2 _ε package interface commands	106
4.13	Deprecated that needs cleanup at some point	109
4.14	Internal commands needed elsewhere	111
	Index	113

1 Introduction

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

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

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

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

2 Package writer interface

The hook management system is offered as a set of CamelCase commands for traditional L^AT_EX 2_ε packages (and for use in the document preamble if needed) as well as `expl3` commands for modern packages, that use the L3 programming layer of L^AT_EX. 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 L^AT_EX 2_ε 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 generic 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. The string `??` can't be used as a hook name because it has a special significance as a placeholder in hook rules.

\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 $\{\langle hook \rangle\}$
----------------------------	---

After this declaration¹ the $\langle hook \rangle$ 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 `ltxcmdhooks-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 $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\ActivateGenericHook	\ActivateGenericHook $\{\langle hook \rangle\}$
-----------------------------	--

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`. If the hook is already activated, this command does nothing.

Note that this command does not undo the effect of `\DisableGenericHook`. See section 2.6 for a discussion of when this declaration is appropriate.

2.1.3 Using hooks in code

Using a hook that is executing the code that has been associated with it is only allowed if the hook has been previously declared with `\NewHook`. For performance reason there are no runtime checks for this and it is the responsibility of the programmer of a package to ensure that all hooks that are used in a package (with one of the commands in this section) are declared first.

\UseHook	\UseHook $\{\langle hook \rangle\}$
-----------------	--

Execute the code stored in the $\langle hook \rangle$.

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 $\langle hook \rangle$ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

\UseHookWithArguments	\UseHookWithArguments $\{\langle hook \rangle\}$ $\{\langle number \rangle\}$ $\{\langle arg_1 \rangle\}$ $\dots$ $\{\langle arg_n \rangle\}$
------------------------------	--

Execute the code stored in the $\langle hook \rangle$ and pass the arguments $\{\langle arg_1 \rangle\}$ through $\{\langle arg_n \rangle\}$ to the $\langle hook \rangle$. Otherwise, it works exactly like `\UseHook`. The $\langle number \rangle$ should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove $\langle number \rangle$ items from the input. Section 2.7 explains hooks with arguments.

The $\langle hook \rangle$ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

¹In 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.

<code>\UseOneTimeHook</code>	<code>\UseOneTimeHook {$\langle hook \rangle$}</code>
------------------------------	--

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 `\langle addto-cmd \rangle` 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 `\langle addto-cmd \rangle` 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 `{ $\langle hook \rangle$ }` 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 `{ $\langle hook \rangle$ }` should be avoided, but if this is done then neither will execute after the first `\UseOneTimeHook`.

The `$\langle hook \rangle$` *cannot* be specified using the dot-syntax. A leading `.` is treated literally. See section 2.1.5 for details.

<code>\UseOneTimeHookWithArguments</code>	<code>\UseOneTimeHookWithArguments {$\langle hook \rangle$} {$\langle number \rangle$} {$\langle arg_1 \rangle$} ... {$\langle arg_n \rangle$}</code>
---	---

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

In contrast to the commands from the previous section, declarations such as `\AddToHook` or `\DeclareHookRule` can be used even when the hook is not yet declared. The rationale is that the hook declaration may be in some package that is loaded later, or perhaps not loaded at all.

A side effect of this design is that misspellings do not raise an error but are simply regarded as declarations for hooks with a different name.

<code>\AddToHook</code>	<code>\AddToHook {<hook>} [<label>] {<code>}</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.

<code>\AddToHookWithArguments</code>	<code>\AddToHookWithArguments {<hook>} [<label>] {<code>}</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.

<code>\RemoveFromHook</code>	<code>\RemoveFromHook {<hook>} [<label>]</code>
------------------------------	---

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.

Important:

The `\RemoveFromHook` command should be only used if one has full control over the code chunk to be removed. In particular it should not be used to remove code chunks from other packages! For this the `voids` relation is provided.

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

Note that you can't cancel the setting with

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

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.

<code>\AddToHookNext</code>	<code>\AddToHookNext {<hook>} {<code>}</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.

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

<code>\AddToHookNextWithArguments</code>	<code>\AddToHookNextWithArguments {<hook>} {<code>}</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.

<code>\ClearHookNext</code>	<code>\ClearHookNext {<hook>}</code>
-----------------------------	--

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 $\langle\textit{default label}\rangle$ 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 $\langle\textit{default label}\rangle$. This may have exceptions—see `\PushDefaultHookLabel`.

This syntax is available in all $\langle\textit{label}\rangle$ arguments and most $\langle\textit{hook}\rangle$ arguments, both in the $\text{\LaTeX} 2_{\epsilon}$ interface, and the $\text{\LaTeX} 3$ interface described in section 2.2.

Note, however, that the replacement of `.` by the $\langle\textit{default label}\rangle$ takes place when the hook command is executed, so actions that are somehow executed after the package ends will have the wrong $\langle\textit{default label}\rangle$ 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 $\langle\textit{label}\rangle$, then the $\langle\textit{default label}\rangle$ can be set using `\PushDefaultHookLabel{...}\PopDefaultHookLabel` or `\SetDefaultHookLabel{...}`.

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

<code>\PushDefaultHookLabel</code>	<code>\PushDefaultHookLabel {⟨default label⟩}</code>
<code>\PopDefaultHookLabel</code>	<code>⟨code⟩</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 L^AT_EX 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 (TikZ’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`.

<code>\SetDefaultHookLabel</code>	<code>\SetDefaultHookLabel {⟨default label⟩}</code>
-----------------------------------	---

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 precautions to determine whether some other package had also been loaded (before or after) and then to find some ways to alter its behavior accordingly. In addition it was often the user's responsibility to load packages in the right order so that alterations made by packages were done in that same order; and in 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 to specify explicitly the order in which they should be processed.

The rules can be declared for hooks before the hook has been declared with `\NewHook` and they are allowed to refer to code labels that do not yet exist, e.g., because a package defining the code chunk with that label has not yet been loaded. When the hook code is finally sorted for fast execution, all rules that apply are acted on and the others are ignored.

This offers the flexibility needed to handle complicated relationships between code from different packages and to set this up beforehand in a way that is independent of whether or not the packages are actually loaded in a specific document. The downside of this is that misspellings of hook names or code labels will not raise any error, instead the rule will simply never apply!

\DeclareHookRule `\DeclareHookRule {<hook>} {<label1>} {<relation>} {<label2>}`

Defines a relation between $\langle label1 \rangle$ and $\langle label2 \rangle$ for a given $\langle hook \rangle$. If $\langle hook \rangle$ is `??` this defines a default relation for all hooks that use the two labels, i.e., that have chunks of code labeled with $\langle label1 \rangle$ and $\langle label2 \rangle$.

Currently, the supported relations are the following:

before or **<** Code for $\langle label1 \rangle$ comes before code for $\langle label2 \rangle$.

after or **>** Code for $\langle label1 \rangle$ comes after code for $\langle label2 \rangle$.

incompatible-warning Only code for either $\langle label1 \rangle$ or $\langle label2 \rangle$ 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-warning** but instead of a warning a L^AT_EX error is raised, and the code for both labels are dropped from that hook until the conflict is resolved.

voids Code for $\langle label1 \rangle$ overwrites code for $\langle label2 \rangle$. More precisely, code for $\langle label2 \rangle$ 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 $\langle label1 \rangle$ and $\langle label2 \rangle$ 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. In all cases rules specific to a given hook take precedence over default rules that use `??` as the $\langle hook \rangle$.

If a default rule is applied, it is done before reversing the label order in a reversed hook, e.g., **before** in a default rule effectively becomes **after** in such a hook. In contrast, a rule for a specific hook is always applied to the state after any reversal (i.e., the state you see when using **\ShowHook** on that hook).

The $\langle hook \rangle$ and $\langle label \rangle$ 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 $\langle label1 \rangle$ and $\langle label2 \rangle$ are unrelated for the given $\langle hook \rangle$.

`\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.)

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. The rationale is that in hook pairs (in which the ordering in one is reversed) default rules have to be reversed too in nearly all scenarios. If this is not the case, a default rule can't be used or has to be overwritten with an explicit `\DeclareHookRule` for that specific hook.

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

The `<label>` can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

2.1.8 Querying hooks

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

- exist and be empty;
- exist and be non-empty; and
- not exist (in which case emptiness doesn't apply);

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

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

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

A hook is said to be empty when no code was added to it, either to its permanent code pool, or to its “next” token list. The hook doesn't need to be declared to have code added to its code pool. A hook is said to exist when it was declared with `\NewHook` or some variant thereof. Generic hooks such as `file` and `env` hooks are automatically declared when code is added to them.

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

<code>\IfHookEmptyTF</code>	★	<code>\IfHookEmptyTF {<hook>} {<true code>} {<false code>}</code>
<code>\IfHookEmptyT</code>	★	Tests if the <code><hook></code> is empty (<i>i.e.</i> , no code was added to it using either <code>\AddToHook</code> or
<code>\IfHookEmptyF</code>	★	<code>\AddToHookNext</code>) or such code was removed again (via <code>\RemoveFromHook</code>), and branches
		to either <code><true code></code> or <code><false code></code> depending on the result.
		The <code><hook></code> <i>cannot</i> be specified using the dot-syntax. A leading <code>.</code> 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.

<code>\ShowHook</code>	<code>\ShowHook {<hook>}</code>
<code>\LogHook</code>	<code>\LogHook {<hook>}</code>

Displays information about the `<hook>` such as

- the code chunks (and their labels) added to it,
- any rules set up to order them,
- the computed order in which the chunks are executed,
- any code executed on the next invocation only.

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

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section [2.1.5](#).

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

```

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

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

`<label> -> <code>`

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

```
Document-level (top-level) code (executed  $\langle first/last \rangle$ ):
->  $\langle top-level code \rangle$ 
```

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

```
->  $\langle next-code \rangle$ 
```

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{ $\langle label \rangle$ }[example-hook]`.

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

```
 $\langle label-1 \rangle | \langle label-2 \rangle$  with  $\langle default? \rangle$  relation  $\langle relation \rangle$ 
```

which means that the $\langle relation \rangle$ applies to $\langle label-1 \rangle$ and $\langle label-2 \rangle$, in that order, as detailed in `\DeclareHookRule`. If the relation is `default` it means that this rule applies to $\langle label-1 \rangle$ and $\langle label-2 \rangle$ 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 \DebugHooksOn ... \DebugHooksOff
\DebugHooksOff
```

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, but it can be helpful in case hooks do not work as expected. See also [2.1.9](#) for commands to inspect individual hooks.

2.2 L3 programming layer (expl3) interfaces

This is a quick summary of the L^AT_EX3 programming interfaces for use with packages written in `expl3`. In contrast to the L^AT_EX2_ε interfaces they always use mandatory arguments only, e.g., you always have to specify the $\langle label \rangle$ for a code chunk. We therefore suggest to use the declarations discussed in the previous section even in `expl3` packages, but the choice is yours.

```
\hook_new:n \hook_new:n { $\langle hook \rangle$ }
\hook_new_reversed:n \hook_new_reversed:n { $\langle hook \rangle$ }
\hook_new_pair:nn \hook_new_pair:nn { $\langle hook-1 \rangle$ } { $\langle hook-2 \rangle$ }
```

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

The $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section [2.1.5](#).

<code>\hook_new_with_args:nn</code>	<code>\hook_new_with_args:nn {\langle hook \rangle} {\langle number \rangle}</code>
<code>\hook_new_reversed_with_args:nn</code>	<code>\hook_new_reversed_with_args:nn {\langle hook \rangle} {\langle number \rangle}</code>
<code>\hook_new_pair_with_args:nnn</code>	<code>\hook_new_pair_with_args:nnn {\langle hook-1 \rangle} {\langle hook-2 \rangle} {\langle number \rangle}</code>

Creates a new $\langle hook \rangle$ with normal or reverse ordering of code chunks, that takes $\langle number \rangle$ arguments from the input stream when it is used. `\hook_new_pair_with_args:nn` creates a pair of such hooks with $\{\langle hook-2 \rangle\}$ being a reversed hook. If a hook name is already taken, an error is raised and the hook is not created.

The $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

<code>\hook_disable_generic:n</code>	<code>\hook_disable_generic:n {\langle hook \rangle}</code>
--------------------------------------	---

Marks $\{\langle hook \rangle\}$ 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 `ltxcmdhooks-doc`) if they receive code.

The $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

<code>\hook_activate_generic:n</code>	<code>\hook_activate_generic:n {\langle hook \rangle}</code>
---------------------------------------	--

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.

<code>\hook_use:n</code>	<code>\hook_use:n {\langle hook \rangle}</code>
<code>\hook_use:nnw</code>	<code>\hook_use:nnw {\langle hook \rangle} {\langle number \rangle} {\langle arg_1 \rangle} \dots {\langle arg_n \rangle}</code>

Executes the $\{\langle hook \rangle\}$ 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 $\langle number \rangle$ should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove $\langle number \rangle$ items from the input.

The $\langle hook \rangle$ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

<code>\hook_use_once:n</code>	<code>\hook_use_once:n {\langle hook \rangle}</code>
<code>\hook_use_once:nnw</code>	<code>\hook_use_once:nnw {\langle hook \rangle} {\langle number \rangle} {\langle arg_1 \rangle} \dots {\langle arg_n \rangle}</code>

Changes the $\{\langle hook \rangle\}$ status so that from now on any addition to the hook code is executed immediately. Then execute any $\{\langle hook \rangle\}$ 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 $\langle number \rangle$ should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove $\langle number \rangle$ items from the input.

The $\langle hook \rangle$ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

<code>\hook_gput_code:nnn</code>	<code>\hook_gput_code:nnn</code>	<code>{\hook} {\label} {\code}</code>
<code>\hook_gput_code_with_args:nnn</code>	<code>\hook_gput_code_with_args:nnn</code>	<code>{\hook} {\label} {\code}</code>

Adds a chunk of $\langle code \rangle$ to the $\langle hook \rangle$ labeled $\langle label \rangle$. If the label already exists the $\langle code \rangle$ is appended to the already existing code.

If `\hook_gput_code_with_args:nnn` is used, the $\langle code \rangle$ 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 $\langle hook \rangle$ (of the kernel or another package) then the convention is to use the package name as the $\langle label \rangle$ not some internal module name or some other arbitrary string.

The $\langle hook \rangle$ and $\langle label \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

<code>\hook_gput_next_code:nn</code>	<code>\hook_gput_next_code:nn</code>	<code>{\hook} {\code}</code>
<code>\hook_gput_next_code_with_args:nn</code>	<code>\hook_gput_next_code_with_args:nn</code>	<code>{\hook} {\code}</code>

Adds a chunk of $\langle code \rangle$ for use only in the next invocation of the $\langle hook \rangle$. Once used it is gone.

If `\hook_gput_next_code_with_args:nn` is used, the $\langle code \rangle$ 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 $\langle code \rangle$.

The $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

<code>\hook_gclear_next_code:n</code>	<code>\hook_gclear_next_code:n</code>	<code>{\hook}</code>
---------------------------------------	---------------------------------------	----------------------

Undo any earlier `\hook_gput_next_code:nn`.

<code>\hook_gremove_code:nn</code>	<code>\hook_gremove_code:nn</code>	<code>{\hook} {\label}</code>
------------------------------------	------------------------------------	-------------------------------

Removes any code for $\langle hook \rangle$ labeled $\langle label \rangle$.

If there is no code under the $\langle label \rangle$ in the $\langle hook \rangle$, or if the $\langle hook \rangle$ 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 $\langle hook \rangle$ and $\langle label \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

<hr/> <hr/>	<code>\hook_gset_rule:nnnn</code>	<code>\hook_gset_rule:nnnn {<hook>} {<label1>} {<relation>} {<label2>}</code>
		Relate <code><label1></code> with <code><label2></code> when used in <code><hook></code> . See <code>\DeclareHookRule</code> for the allowed <code><relation></code> s. If <code><hook></code> is ?? a default rule is specified. The <code><hook></code> and <code><label></code> 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 <code><label></code> arguments, but it usually makes sense to be used in only one of them.
<hr/> <hr/>	<code>\hook_if_empty_p:n *</code> <code>\hook_if_empty:nTF *</code>	<code>\hook_if_empty:nTF {<hook>} {<true code>} {<false code>}</code> Tests if the <code><hook></code> is empty (<i>i.e.</i> , no code was added to it using either <code>\AddToHook</code> or <code>\AddToHookNext</code>), and branches to either <code><true code></code> or <code><false code></code> depending on the result. The <code><hook></code> <i>cannot</i> be specified using the dot-syntax. A leading . is treated literally.
<hr/> <hr/>	<code>\hook_show:n</code> <code>\hook_log:n</code>	<code>\hook_show:n {<hook>}</code> <code>\hook_log:n {<hook>}</code> Displays information about the <code><hook></code> such as <ul style="list-style-type: none"> • 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. <code>\hook_log:n</code> prints the information to the .log file, and <code>\hook_show:n</code> prints them to the terminal/command window and starts T _E X's prompt (only if <code>\errorstopmode</code>) to wait for user action. The <code><hook></code> can be specified using the dot-syntax to denote the current package name. See section 2.1.5.
<hr/> <hr/>	<code>\hook_debug_on:</code> <code>\hook_debug_off:</code>	<code>\hook_debug_on:</code> 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 example⁴, suppose there is a package adding the following:

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

As a result, all quotes will be in italics. Now suppose further that another `package-too` makes the quotes also in blue and therefore adds:

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

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

```
package-1, package-too
```

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

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

and an error message 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.
```

If there is a matching default rule (done with `\DeclareDefaultHookRule` or with `??` for the hook name) then this default rule is applied before the reversal so that the order in the reversed hook mirrors the one in the normal hook. However, all rules specific to a hook happen always after the reversal of the execution order, 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.

⁴There are simpler ways to achieve the same effect.

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 hooks⁵ (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 $\langle \# \text{ of invocations} \rangle - 1$ 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.

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

The generic hooks provided by L^AT_EX 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 T_EX's limit of 9 arguments applies, and a low-level T_EX 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. Behavior is unpredictable⁶ 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:

```
\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

⁶The 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.

references more arguments than will be declared for the hook, there will be a low-level $\text{\TeX}$ error about an “Illegal parameter number” at the time the hook is declared, which will be hard to track down because at that point $\text{\TeX}$ 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 $\text{\LaTeX}$ kernel hooks

There are a few places where it is absolutely essential for $\text{\LaTeX}$ 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 $\text{\LaTeX}$).

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 $\text{\LaTeX 2}_\epsilon$ interfaces

$\text{\LaTeX 2}_\epsilon$ 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 $\text{\LaTeX}$ and more will follow. See the next section for what is already available.

⁷As with everything in $\text{\TeX}$ 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!”

<code>\AtBeginDocument</code>	<code>\AtBeginDocument [<i>label</i>] {<i>code</i>}</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.

<code>\AtEndDocument</code>	<code>\AtEndDocument [<i>label</i>] {<i>code</i>}</code>
-----------------------------	--

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

The few hooks that existed previously in $\text{\LaTeX 2}_\epsilon$ 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 $\text{\LaTeX}$ we will turn this legacy support off, as it does unnecessary slow down the processing.

3 $\text{\LaTeX 2}_\epsilon$ commands and environments augmented by hooks

In this section we describe the standard hooks that are now offered by $\text{\LaTeX}$, 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 `ltxcmdhooks-doc.pdf` or `ltxfilehook-doc.pdf`.

The generic hooks provided by $\text{\LaTeX}$ belong to one of the six types:

- env** Hooks executed before and after environments – $\langle name \rangle$ is the name of the environment, and available values for $\langle position \rangle$ are **before**, **begin**, **end**, and **after**;
- cmd** Hooks added to and executed before and after commands – $\langle name \rangle$ is the name of the command, and available values for $\langle position \rangle$ are **before** and **after**;
- file** Hooks executed before and after reading a file – $\langle name \rangle$ is the name of the file (with extension), and available values for $\langle position \rangle$ are **before** and **after**;
- package** Hooks executed before and after loading packages – $\langle name \rangle$ is the name of the package, and available values for $\langle position \rangle$ are **before** and **after**;
- class** Hooks executed before and after loading classes – $\langle name \rangle$ is the name of the class, and available values for $\langle position \rangle$ are **before** and **after**;
- include** Hooks executed before and after `\included` files – $\langle name \rangle$ is the name of the included file (without the `.tex` extension), and available values for $\langle position \rangle$ 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 $\langle env \rangle$ has now four associated hooks coming with it:

- env/ $\langle env \rangle$ /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/ $\langle env \rangle$ /begin** This hook is executed as part of `\begin` directly in front of the code specific to the environment start (e.g., the third argument of `\NewDocumentEnvironment` and the second argument of `\newenvironment`). Its scope is the environment body.
- env/ $\langle env \rangle$ /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 forth argument of `\NewDocumentEnvironment` and the third argument of `\newenvironment`).
- env/ $\langle env \rangle$ /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/ $\langle env \rangle$ /before** and to **env/ $\langle env \rangle$ /after** they can add surrounding environments and the order of closing them happens in the right sequence.

Given that these generic hook names involve `/` as part of their name they would not work if one tries to define an environment using a name that involves a `/`.⁸

Generic environment hooks are never one-time hooks even with environments that are supposed to appear only once in a document.⁹ In contrast to other hooks there is also no need to declare them using `\NewHook`.

⁸Officially, L^AT_EX names for environments should only consist of a sequence of letters, numbers, and the character `*`, i.e., this is not a new restriction.

⁹Thus 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.

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 `\endenv` (e.g., to avoid the environment grouping) they are not available. If you want them available in code using this method, you would need to add them yourself, i.e., write something like

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

to add the outer hooks, etc.

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

<hr/> <hr/>	<code>\BeforeBeginEnvironment</code>	<code>\BeforeBeginEnvironment</code> [<i>label</i>] { <i>env</i> } { <i>code</i> }	This declaration adds to the <code>env/⟨env⟩/before</code> hook using the <i>label</i> . If <i>label</i> is not given, the <i>default label</i> is used (see section 2.1.5).
<hr/> <hr/>	<code>\AtBeginEnvironment</code>	<code>\AtBeginEnvironment</code> [<i>label</i>] { <i>env</i> } { <i>code</i> }	This is like <code>\BeforeBeginEnvironment</code> but it adds to the <code>env/⟨env⟩/begin</code> hook.
<hr/> <hr/>	<code>\AtEndEnvironment</code>	<code>\AtEndEnvironment</code> [<i>label</i>] { <i>env</i> } { <i>code</i> }	This is like <code>\BeforeBeginEnvironment</code> but it adds to the <code>env/⟨env⟩/end</code> hook.
<hr/> <hr/>	<code>\AfterEndEnvironment</code>	<code>\AfterEndEnvironment</code> [<i>label</i>] { <i>env</i> } { <i>code</i> }	This is like <code>\BeforeBeginEnvironment</code> but it adds to the <code>env/⟨env⟩/after</code> hook.

3.1.2 Generic hooks for commands

Similar to environments there are now (at least in theory) two generic hooks available for any $\text{\LaTeX}$ 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 `ltxcmdhooks-doc.pdf` or with code in `ltxcmdhooks-code.pdf`.

3.1.3 Generic hooks provided by file loading operations

There are several hooks added to $\text{\LaTeX}$'s process of loading file via its high-level interfaces such as `\input`, `\include`, `\usepackage`, `\RequirePackage`, etc. These are documented in `ltxfilehook-doc.pdf` or with code in `ltxfilehook-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 L^AT_EX'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}`

L^AT_EX 2_ε 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 L^AT_EX 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). Just before this hook `\write` is redefined and is now always `\immediate\write`. This means that writing e.g. a label or something to the `.toc` works despite the fact that no more pages are output. 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, $\text{\LaTeX}$ 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 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 $\text{\LaTeX}$ has finished the document processing.

3.4 Hooks provided by `\shipout` operations

There are several hooks and mechanisms added to $\text{\LaTeX}$'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 metafamilies (rm/sf/tt) and the metaseries (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 metafamilies (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

```

1 <@@=hook>
2 <*2ekernel | latexrelease>
3 \ExplSyntaxOn
4 <latexrelease>\NewModuleRelease{2020/10/01}{lthooks}
5 <latexrelease>                                {The-hook-management-system}

```

4.1 Debugging

```

\g__hook_debug_bool Holds the current debugging state.
6 \bool_new:N \g__hook_debug_bool

(End of definition for \g__hook_debug_bool.)

\hook_debug_on: Turns debugging on and off by redefining \__hook_debug:n.
\hook_debug_off:
\__hook_debug:n
\__hook_debug_gset:
7 \cs_new_eq:NN \__hook_debug:n \use_none:n
8 \cs_new_protected:Npn \hook_debug_on:
9 {
10   \bool_gset_true:N \g__hook_debug_bool
11   \__hook_debug_gset:
12 }
13 \cs_new_protected:Npn \hook_debug_off:
14 {
15   \bool_gset_false:N \g__hook_debug_bool
16   \__hook_debug_gset:
17 }
18 \cs_new_protected:Npn \__hook_debug_gset:
19 {
20   \cs_gset_protected:Npe \__hook_debug:n ##1
21   { \bool_if:NT \g__hook_debug_bool {##1} }
22 }

```

(End of definition for `\hook_debug_on:` and others. These functions are documented on page 18.)

4.2 Borrowing from internals of other kernel modules

`\__hook_str_compare:nn` Private copy of `\__str_if_eq:nn`
`23 \cs_new_eq:NN \__hook_str_compare:nn \__str_if_eq:nn`
(End of definition for __hook_str_compare:nn.)

4.3 Declarations

`\l__hook_tmpa_bool` Scratch boolean used throughout the package.
`24 \bool_new:N \l__hook_tmpa_bool`
(End of definition for \l__hook_tmpa_bool.)

`\l__hook_return_tl` Scratch variables used throughout the package.
`\l__hook_tmpa_tl` `25 \tl_new:N \l__hook_return_tl`
`\l__hook_tmpb_tl` `26 \tl_new:N \l__hook_tmpa_tl`
`27 \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.
`28 \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.
`29 \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.
`30 \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).
`31 \prop_new:N \g__hook_used_prop`
(End of definition for \g__hook_used_prop.)

`\g__hook_hook_curr_name_tl` Default label used for hook commands, and a stack to keep track of packages within
`\g__hook_name_stack_seq` packages.
`32 \tl_new:N \g__hook_hook_curr_name_tl`
`33 \seq_new:N \g__hook_name_stack_seq`
(End of definition for \g__hook_hook_curr_name_tl and \g__hook_name_stack_seq.)

`\__hook_tmp:w` Temporary macro for generic usage.
`34 \cs_new_eq:NN \__hook_tmp:w ?`
(End of definition for __hook_tmp:w.)

`\c__hook_empty_tl` An empty token list, and one containing nine parameters.

`\c__hook_nine_parameters_tl`

```

35 \tl_const:Nn \c__hook_empty_tl { }
36 \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`.)

`\str_count:e` Some variants of `expl3` functions.

FMi: should probably be moved to `expl3`

```

37 \cs_generate_variant:Nn \str_count:n { e }

```

(End of definition for `\str_count:e`.)

`\s__hook_mark` Scan mark used for delimited arguments.

```

38 \scan_new:N \s__hook_mark

```

(End of definition for `\s__hook_mark`.)

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

`\__hook_use_i_delimit_by_s_mark:nw`

```

39 \cs_new:Npn \__hook_use_none_delimit_by_s_mark:w #1 \s__hook_mark { }
40 \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`.

```

41 \cs_new_protected:Npn \__hook_tl_set:cn #1#2
42 { \cs_set_nopar:cpe {#1} { \__kernel_exp_not:w {#2} } }

```

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

`\__hook_tl_gset:Nn` Same as above.

`\__hook_tl_gset:Ne`

`\__hook_tl_gset:cn`

`\__hook_tl_gset:co`

`\__hook_tl_gset:ce`

```

43 \cs_new_protected:Npn \__hook_tl_gset:Nn #1#2
44 { \cs_gset_nopar:Npe #1 { \__kernel_exp_not:w {#2} } }
45 \cs_new_protected:Npn \__hook_tl_gset:Ne #1#2
46 { \cs_gset_nopar:Npe #1 {#2} }
47 \cs_generate_variant:Nn \__hook_tl_gset:Nn { c, co }
48 \cs_generate_variant:Nn \__hook_tl_gset:Ne { c }

```

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

`\__hook_tl_gput_right:Nn` Same as above.

`\__hook_tl_gput_right:Ne`

`\__hook_tl_gput_right:cn`

```

49 \cs_new_protected:Npn \__hook_tl_gput_right:Nn #1#2
50 { \__hook_tl_gset:Ne #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
51 \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.

```

52 \cs_new_protected:Npn \__hook_tl_gput_left:Nn #1#2
53 {
54   \__hook_tl_gset:Ne #1
55   { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
56 }

```

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

`\__hook_tl_gset_eq:NN` Same as above.

```

57 \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` Same as above.
`\__hook_tl_gclear:c`

```

58 \cs_new_protected:Npn \__hook_tl_gclear:N #1
59 { \__hook_tl_gset_eq:NN #1 \c_empty_tl }
60 \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_⟨hook⟩_code_prop` Hooks have a name (called `⟨hook⟩` in the description below) and for each hook we have
`\__hook_⟨hook⟩` to provide a number of data structures. These are

`\g__hook_⟨hook⟩_reversed_tl` `\g__hook_⟨hook⟩_code_prop` A property list holding the code for the hook in separate
`\g__hook_⟨hook⟩_declared_tl` chunks. The keys are by default the package names that add code to the hook, but
`\g__hook_⟨hook⟩_parameter_tl` it is possible for packages to define other keys.
`\__hook_next_⟨hook⟩`
`\__hook_toplevel_⟨hook⟩` `\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 in 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 if 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 $\langle name \rangle$ as its argument, e.g., `begindocument`.

```

61 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_new_with_args:nn}
62 <latexrelease>{Hooks~with~args}
63 \cs_new_protected:Npn \hook_new:n #1
64 { \__hook_normalize_hook_args:Nn \__hook_new:nn {#1} { 0 } }
65 \cs_new_protected:Npn \hook_new_with_args:nn #1 #2
66 { \__hook_normalize_hook_args:Nn \__hook_new:nn {#1} {#2} }
67 \cs_new_protected:Npn \__hook_new:nn #1 #2
68 {

```

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.

```

69 \__hook_if_declared:nTF {#1}
70 { \msg_error:nnn { hooks } { exists } {#1} }

```

```

71     {
72       \tl_new:c { g__hook_#1_declared_tl }
73       \cs_undefine:c { __hook~#1 }
74       \cs_undefine:c { c__hook_#1_parameter_tl }
75       \__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`).

```

76       \__hook_update_hook_code:n {#1}
77     }
78   }
79 <latexrelease>\EndIncludeInRelease
80 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_new_with_args:nn}
81 <latexrelease>      {Hooks~with~args}
82 <latexrelease>\cs_gset_protected:Npn \hook_new:n #1
83 <latexrelease>  { \__hook_normalize_hook_args:Nn \__hook_new:n {#1} }
84 <latexrelease>\cs_undefine:N \__hook_new:nn
85 <latexrelease>\cs_gset_protected:Npn \__hook_new:n #1
86 <latexrelease>  {
87 <latexrelease>    \__hook_if_declared:nTF {#1}
88 <latexrelease>    { \msg_error:nnn { hooks } { exists } {#1} }
89 <latexrelease>    {
90 <latexrelease>      \tl_new:c { g__hook_#1_declared_tl }
91 <latexrelease>      \__hook_make_usable:n {#1}
92 <latexrelease>    }
93 <latexrelease>  }
94 <latexrelease>\cs_gset_protected:Npn \hook_new_with_args:nn #1 { }
95 <latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_new:n`, `\hook_new_with_args:nn`, and `\__hook_new:nn`. These functions are documented on page 15.)

`\__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.

```

96 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_make_usable:nn}
97 <latexrelease>      {Hooks~with~args}
98 \cs_new_protected:Npn \__hook_make_usable:nn #1 #2
99   {

```

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.

```

100   \__hook_if_usable:nF {#1}
101   {
102     \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`).

```

103     \__kernel_cs_parm_from_arg_count:nnF
104     { \tl_const:cn { c__hook_#1_parameter_tl } } {#2}
105     {
106       \msg_error:nnnn { hooks } { too-many-args } {#1} {#2}

```

```

107         \tl_const:ce { c__hook_#1_parameter_tl }
108         { \exp_not:V \c__hook_nine_parameters_tl }
109     }

```

After that, use `\__hook_normalize_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_normalize_cs_args:nn` does the normalization by using the `\c__hook_<hook>_parameter_tl` defined just above.

```

110     \__hook_normalize_cs_args:nn { _toplevel } {#1}
111     \__hook_normalize_cs_args:nn { _next } {#1}

```

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

```

112     \__hook_code_gset:nn {#1} { }

```

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

```

113     \__hook_init_structure:n {#1}

```

The call to `\__hook_normalize_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`.

```

114     \__hook_normalize_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.

```

115     \clist_if_exist:cF { g__hook_#1_labels_clist }
116     {
117         \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.

```

118         \tl_new:c { g__hook_#1_reversed_tl }
119     }

```

The above is all in L3 convention, but we also provide an interface to legacy `LATEX 2ε` 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!

```

120     \__hook_include_legacy_code_chunk:n {#1}
121 }
122 }
123 <latexrelease>\EndIncludeInRelease

124 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_make_usable:nn}
125 <latexrelease>{Hooks~with~args}
126 <latexrelease>\cs_undefine:N \__hook_make_usable:nn
127 <latexrelease>\cs_gset_protected:Npn \__hook_make_usable:n #1
128 <latexrelease> {
129 <latexrelease>     \tl_if_exist:cF { __hook~#1 }
130 <latexrelease>     {

```

```

131 <latexrelease> \seq_gput_right:Nn \g__hook_all_seq {#1}
132 <latexrelease> \tl_new:c { __hook~#1 }
133 <latexrelease> \__hook_init_structure:n {#1}
134 <latexrelease> \clist_new:c { g__hook_#1_labels_clist }
135 <latexrelease> \tl_new:c { g__hook_#1_reversed_tl }
136 <latexrelease> \__hook_include_legacy_code_chunk:n {#1}
137 <latexrelease> }
138 <latexrelease> }
139 <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`.

```

140 <latexrelease> \IncludeInRelease{2023/06/01}{\__hook_init_structure:n}
141 <latexrelease> {Hooks~with~args}
142 \cs_new_protected:Npn \__hook_init_structure:n #1
143 {
144   \__hook_if_structure_exist:nF {#1}
145   {
146     \prop_new:c { g__hook_#1_code_prop }
147     \__hook_toplevel_gset:nn {#1} { }
148     \__hook_next_gset:nn {#1} { }
149   }
150 }
151 <latexrelease> \EndIncludeInRelease

152 <latexrelease> \IncludeInRelease{2020/10/01}{\__hook_init_structure:n}
153 <latexrelease> {Hooks~with~args}
154 <latexrelease> \cs_gset_protected:Npn \__hook_init_structure:n #1
155 <latexrelease> {
156 <latexrelease>   \__hook_if_structure_exist:nF {#1}
157 <latexrelease>   {
158 <latexrelease>     \prop_new:c { g__hook_#1_code_prop }
159 <latexrelease>     \tl_new:c { __hook_toplevel~#1 }
160 <latexrelease>     \tl_new:c { __hook_next~#1 }
161 <latexrelease>   }
162 <latexrelease> }
163 <latexrelease> \EndIncludeInRelease

```

(End of definition for __hook_init_structure:n.)

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

```

\hook_new_reversed_with_args:nn
\__hook_new_reversed:nn
164 <latexrelease> \IncludeInRelease{2023/06/01}{\hook_new_reversed_with_args:nn}
165 <latexrelease> {Hooks~with~args}
166 \cs_new_protected:Npn \hook_new_reversed:n #1
167 { \__hook_normalize_hook_args:Nn \__hook_new_reversed:nn {#1} { 0 } }

```

```

168 \cs_new_protected:Npn \hook_new_reversed_with_args:nn #1 #2
169 { \__hook_normalize_hook_args:Nn \__hook_new_reversed:nn {#1} {#2} }
170 \cs_new_protected:Npn \__hook_new_reversed:nn #1 #2
171 {
172   \__hook_if_declared:nTF {#1}
173   { \msg_error:nnn { hooks } { exists } {#1} }
174   {
175     \__hook_new:nn {#1} {#2}
176     \tl_gset:cn { g__hook_#1_reversed_tl } { - }
177   }
178 }
179 <latexrelease>\EndIncludeInRelease

180 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_new_reversed_with_args:nn}
181 <latexrelease> {Hooks~with~args}
182 <latexrelease>\cs_gset_protected:Npn \hook_new_reversed:n #1
183 <latexrelease> { \__hook_normalize_hook_args:Nn \__hook_new_reversed:n {#1} }
184 <latexrelease>\cs_undefine:N \__hook_new_reversed:nn
185 <latexrelease>\cs_gset_protected:Npn \__hook_new_reversed:n #1
186 <latexrelease> {
187   \__hook_new:n {#1}
188   \tl_gset:cn { g__hook_#1_reversed_tl } { - }
189 <latexrelease> }
190 <latexrelease>\cs_undefine:N \__hook_new_reversed:nn
191 <latexrelease>\cs_gset_protected:Npn \hook_new_reversed_with_args:nn #1 #2 { }
192 <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 15.)

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

```

\hook_new_pair_with_args:nnn
193 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_new_pair_with_args:nnn}
194 <latexrelease> {Hooks~with~args}
195 \cs_new_protected:Npn \hook_new_pair:nn #1#2
196 { \__hook_normalize_hook_args:Nnn \__hook_new_pair:nnn {#1} {#2} { 0 } }
197 \cs_new_protected:Npn \hook_new_pair_with_args:nnn #1#2#3
198 { \__hook_normalize_hook_args:Nnn \__hook_new_pair:nnn {#1} {#2} {#3} }
199 \cs_new_protected:Npn \__hook_new_pair:nnn #1 #2 #3
200 {
201   \__hook_if_declared:nTF {#1}
202   { \msg_error:nnn { hooks } { exists } {#1} }
203   {
204     \__hook_if_declared:nTF {#2}
205     { \msg_error:nnn { hooks } { exists } {#2} }
206     {
207       \__hook_new:nn {#1} {#3}
208       \__hook_new_reversed:nn {#2} {#3}
209     }
210   }
211 }
212 <latexrelease>\EndIncludeInRelease

213 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_new_pair_with_args:nnn}
214 <latexrelease> {Hooks~with~args}
215 <latexrelease>\cs_gset_protected:Npn \hook_new_pair:nn #1#2
216 <latexrelease> {

```

```

217 <latexrelease> \hook_new:n {#1}
218 <latexrelease> \hook_new_reversed:n {#2}
219 <latexrelease> }
220 <latexrelease> \cs_gset_protected:Npn \hook_new_pair_with_args:nnn #1#2#3
221 <latexrelease> { }
222 <latexrelease> \EndIncludeInRelease

```

(End of definition for `\hook_new_pair:nn` and `\hook_new_pair_with_args:nnn`. These functions are documented on page 15.)

`\_hook_include_legacy_code_chunk:n`

The L^AT_EX 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!

```

223 <latexrelease> \IncludeInRelease{2023/06/01}{\_hook_include_legacy_code_chunk:n}
224 <latexrelease> {Hooks~with~args}
225 \cs_new_protected:Npn \_hook_include_legacy_code_chunk:n #1
226 {

```

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

```

227 \tl_if_exist:cT { @#1hook }
228 {

```

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

```

229 \tl_if_empty:cF { @#1hook }
230 {

```

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

```

231 \_hook_replacing_args_false:
232 \use:e
233 {
234 \_hook_hook_gput_code_do:nnn {#1} { legacy }
235 { \exp_not:v { @#1hook } }
236 }
237 \_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.

```

238 \_hook_tl_gclear:c { @#1hook }
239 }
240 }
241 }
242 <latexrelease> \EndIncludeInRelease
243 <latexrelease> \IncludeInRelease{2020/10/01}{\_hook_include_legacy_code_chunk:n}
244 <latexrelease> {Hooks~with~args}
245 <latexrelease> \cs_gset_protected:Npn \_hook_include_legacy_code_chunk:n #1
246 <latexrelease> {
247 <latexrelease> \tl_if_exist:cT { @#1hook }
248 <latexrelease> {
249 <latexrelease> \tl_if_empty:cF { @#1hook }

```

```

250 <latexrelease>          {
251 <latexrelease>          \exp_args:Nnnv \__hook_hook_gput_code_do:nnn
252 <latexrelease>          {#1} { legacy } { @#1hook }
253 <latexrelease>          \__hook_tl_gclear:c { @#1hook }
254 <latexrelease>          }
255 <latexrelease>      }
256 <latexrelease>  }
257 <latexrelease>\EndIncludeInRelease

```

(End of definition for __hook_include_legacy_code_chunk:n.)

4.4.4 Disabling and providing hooks

\hook_disable_generic:n 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.

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

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.

```

258 <latexrelease>\IncludeInRelease{2021/06/01}{\hook_disable_generic:n}
259 <latexrelease>          {Disable~hooks}

260 \cs_new_protected:Npn \hook_disable_generic:n #1
261 { \__hook_normalize_hook_args:Nn \__hook_disable:n {#1} }
262 \cs_new_protected:Npn \__hook_disable:n #1
263 {
264   \tl_gclear_new:c { g__hook_#1_declared_tl }
265   \cs_undefine:c { __hook-#1 }
266 }
267 \prg_new_conditional:Npnn \__hook_if_disabled:n #1 { p, T, F, TF }
268 {
269   \bool_lazy_and:nnTF
270     { \tl_if_exist_p:c { g__hook_#1_declared_tl } }
271     { ! \cs_if_exist_p:c { __hook-#1 } }
272     { \prg_return_true: }
273     { \prg_return_false: }
274 }
275 <latexrelease>\EndIncludeInRelease

276 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_disable_generic:n}
277 <latexrelease>          {Disable~hooks}
278 <latexrelease>

279 <latexrelease>\cs_new_protected:Npn \hook_disable_generic:n #1 {}
280 <latexrelease>
281 <latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_disable_generic:n`, `\__hook_disable:n`, and `\__hook_if_disabled:nTF`. This function is documented on page 16.)

\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.

`\__hook_activate_generic:n`

```

282 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_activate_generic:n}
283 <latexrelease>          {Providing~hooks}

```

```

284 \cs_new_protected:Npn \hook_activate_generic:n #1
285 { \__hook_normalize_hook_args:Nn \__hook_activate_generic:nn {#1} { } }
286 \cs_new_protected:Npn \__hook_activate_generic:nn #1 #2
287 {

```

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

```

288 \__hook_if_disabled:nTF {#1}
289 { \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.

```

290 {
291 \__hook_if_declared:nF {#1}
292 {
293 \tl_new:c { g__hook_#1_declared_tl }
294 \__hook_make_usable:nn {#1} { 0 }
295 \tl_gset:ce { g__hook_#1_reversed_tl }
296 { \__hook_if_generic_reversed:nT {#1} { - } }

```

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

```

297 \__hook_update_hook_code:n {#1}
298 }
299 }
300 }
301 <latexrelease>\EndIncludeInRelease
302 <latexrelease>\IncludeInRelease{2021/06/01}{\hook_activate_generic:n}
303 <latexrelease> {Providing-hooks}
304 <latexrelease>\cs_gset_protected:Npn \__hook_activate_generic:nn #1 #2
305 <latexrelease> {
306 <latexrelease> \__hook_if_disabled:nTF {#1}
307 <latexrelease> { \msg_warning:nnn { hooks } { activate-disabled } {#1} }
308 <latexrelease> {
309 <latexrelease> \__hook_if_declared:nF {#1}
310 <latexrelease> {
311 <latexrelease> \tl_new:c { g__hook_#1_declared_tl }
312 <latexrelease> \__hook_make_usable:n {#1}
313 <latexrelease> \tl_gset:cx { g__hook_#1_reversed_tl }
314 <latexrelease> { \__hook_if_generic_reversed:nT {#1} { - } }
315 <latexrelease> \__hook_update_hook_code:n {#1}
316 <latexrelease> }
317 <latexrelease> }
318 <latexrelease> }
319 <latexrelease>\EndIncludeInRelease
320 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_activate_generic:n}
321 <latexrelease> {Providing-hooks}
322 <latexrelease>\cs_gset_protected:Npn \hook_activate_generic:n #1 { }
323 <latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_activate_generic:n` and `\__hook_activate_generic:n`. This function is documented on page 16.)

4.5 Parsing a label

`\_hook_parse_label_default:nN` 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:.` #2 is a boolean representing if #1 is a label name.

```

324 \cs_new:Npn \_hook_parse_label_default:nN #1#2
325 {
326   \tl_if_novalue:nTF {#1}
327   { \_hook_currname_or_default: }
328   { \tl_trim_spaces_apply:nN {#1} \_hook_parse_dot_label:nN #2 }
329 }
```

(End of definition for `\_hook_parse_label_default:nN`.)

`\_hook_parse_dot_label:nN` Start by checking if the label is empty, which raises an error, and uses the fallback value.
`\_hook_parse_dot_label:w` If not, split the label at a `./`, if any, and check if no tokens are before the `./`, or if the
`\_hook_parse_dot_label_cleanup:w` only character is a `..` If these requirements are fulfilled, the leading `.` is replaced with
`\_hook_parse_dot_label_aux:w` `\_hook_currname_or_default:.` Otherwise the label is returned unchanged. #2 is a boolean representing if #1 is a label name.

```

330 \cs_new:Npn \_hook_parse_dot_label:nN #1#2
331 {
332   \tl_if_empty:nTF {#1}
333   {
334     \bool_if:NTF #2
335     { \msg_expandable_error:nn { hooks } { empty-label } }
336     { \msg_expandable_error:nn { hooks } { empty-hook } }
337     \_hook_currname_or_default:
338   }
339   {
340     \str_if_eq:nnTF {#1} { . }
341     { \_hook_currname_or_default: }
342     { \_hook_parse_dot_label:w #1 ./ \s_hook_mark }
343   }
344 }
345 \cs_new:Npn \_hook_parse_dot_label:w #1 ./ #2 \s_hook_mark
346 {
347   \tl_if_empty:nTF {#1}
348   { \_hook_parse_dot_label_aux:w #2 \s_hook_mark }
349   {
350     \tl_if_empty:nTF {#2}
351     { \_hook_make_name:n {#1} }
352     { \_hook_parse_dot_label_cleanup:w #1 ./ #2 \s_hook_mark }
353   }
354 }
355 \cs_new:Npn \_hook_parse_dot_label_cleanup:w #1 ./ \s_hook_mark {#1}
356 \cs_new:Npn \_hook_parse_dot_label_aux:w #1 ./ \s_hook_mark
357 { \_hook_currname_or_default: / \_hook_make_name:n {#1} }
```

(End of definition for `\_hook_parse_dot_label:nN` 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`.

```

358 \cs_new:Npn \_hook_currname_or_default:
359 {
```

```

360 \tl_if_empty:NTF \g__hook_hook_curr_name_tl
361 {
362   \tl_if_empty:NTF \@currname
363   {
364     \msg_expandable_error:nnn { latex2e } { should-not-happen }
365     { Empty~default~label. }
366     \__hook_make_name:n { label-missing }
367   }
368   { \@currname }
369 }
370 { \g__hook_hook_curr_name_tl }
371 }

```

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

`\__hook_make_name:n` 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.

```

372 \cs_new:Npn \__hook_make_name:n #1
373 {
374   \exp_after:wN \exp_after:wN \exp_after:wN \__hook_make_name:w
375   \exp_after:wN \token_to_str:N \cs:w __hook~ #1 \cs_end:
376 }
377 \exp_last_unbraced:NNNN
378 \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` 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!`

```

379 \cs_new_protected:Npn \__hook_normalize_hook_args_aux:Nn #1 #2
380 {
381   \group_begin:
382   \use:e
383   {
384     \group_end:
385     \exp_not:N #1 #2
386   }
387 }
388 \cs_new_protected:Npn \__hook_normalize_hook_args:Nn #1 #2
389 {
390   \__hook_normalize_hook_args_aux:Nn #1
391   { { \__hook_parse_label_default:nN {#2} \c_false_bool } }
392 }
393 \cs_new_protected:Npn \__hook_normalize_hook_args:Nnn #1 #2 #3
394 {
395   \__hook_normalize_hook_args_aux:Nn #1
396   {
397     { \__hook_parse_label_default:nN {#2} \c_false_bool }

```

```

398         { \_hook_parse_label_default:nN {#3} \c_true_bool }
399     }
400 }
401 \cs_new_protected:Npn \_hook_normalize_hook_rule_args:Nnnnn #1 #2 #3 #4 #5
402 {
403     \_hook_normalize_hook_args_aux:Nn #1
404     {
405         { \_hook_parse_label_default:nN {#2} \c_false_bool }
406         { \_hook_parse_label_default:nN {#3} \c_true_bool }
407         { \tl_trim_spaces:n {#4} }
408         { \_hook_parse_label_default:nN {#5} \c_true_bool }
409     }
410 }

```

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

<pre> _hook_curr_name_push:n _hook_curr_name_push_aux:n _hook_curr_name_pop: _hook_end_document_label_check: </pre>	<p>The token list <code>\g__hook_hook_curr_name_tl</code> 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 <code>\SetDefaultHookLabel</code> to change the default label (in which case <code>\@currname</code> is used).</p>
---	---

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`:

```

411 \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:

```

412 <latexrelease>\seq_if_empty:NT \g__hook_name_stack_seq
413 <latexrelease> { \seq_gput_right:Nn \g__hook_name_stack_seq { top-level } }

```

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

```

414 <latexrelease>\cs_set_protected:Npn \_hook_tmp:w #1 #2 #3
415 <latexrelease> {
416 <latexrelease>     \quark_if_recursion_tail_stop:n {#1}
417 <latexrelease>     \seq_gput_right:Nn \g__hook_name_stack_seq {#1}
418 <latexrelease>     \_hook_tmp:w
419 <latexrelease> }
420 <latexrelease>\exp_after:wN \_hook_tmp:w \@currnamestack
421 <latexrelease> \q_recursion_tail \q_recursion_tail
422 <latexrelease> \q_recursion_tail \q_recursion_stop

```

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

```

423 <latexrelease>\tl_gset:Nx \g__hook_hook_curr_name_tl { \@currname }
424 <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):

```

425 \cs_new_protected:Npn \_hook_curr_name_push:n #1
426 {
427     \exp_args:Ne \_hook_curr_name_push_aux:n
428     {
429         \tl_if_blank:nF {#1}

```

```

430         { \_hook_parse_dot_label:nN {#1} \c_false_bool }
431     }
432 }
433 \cs_new_protected:Npn \_hook_curr_name_push_aux:n #1
434 {
435     \tl_if_blank:nTF {#1}
436     { \msg_error:nn { hooks } { no-default-label } }
437     {
438         \str_if_eq:nnTF {#1} { top-level }
439         {
440             \msg_error:nnnnn { hooks } { set-top-level }
441             { to } { PushDefaultHookLabel } {#1}
442         }
443         {
444             \seq_gpush:NV \g__hook_name_stack_seq \g__hook_hook_curr_name_tl
445             \tl_gset:Nn \g__hook_hook_curr_name_tl {#1}
446         }
447     }
448 }

```

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:

```

449 \cs_new_protected:Npn \_hook_curr_name_pop:
450 {
451     \seq_gpop:NNTF \g__hook_name_stack_seq \l__hook_return_tl
452     { \tl_gset_eq:NN \g__hook_hook_curr_name_tl \l__hook_return_tl }
453     { \msg_error:nn { hooks } { extra-pop-label } }
454 }

```

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):

```

455 \tl_gput_right:Nn \@kernel@after@enddocument@afterlastpage
456 { \_hook_end_document_label_check: }
457 \cs_new_protected:Npn \_hook_end_document_label_check:
458 {
459     \seq_gpop:NNT \g__hook_name_stack_seq \l__hook_return_tl
460     {
461         \msg_error:nne { hooks } { missing-pop-label }
462         { \g__hook_hook_curr_name_tl }
463         \tl_gset_eq:NN \g__hook_hook_curr_name_tl \l__hook_return_tl
464         \_hook_end_document_label_check:
465     }
466 }

```

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

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

```

467 \cs_new_protected:Npn \_hook_set_default_hook_label:n #1
468 {
469     \seq_if_empty:NNTF \g__hook_name_stack_seq
470     {
471         \msg_error:nnnnn { hooks } { set-top-level }

```

`\_hook_set_default_hook_label:n`

```

472         { for } { SetDefaultHookLabel } {#1}
473     }
474     {
475         \exp_args:Ne \__hook_set_default_label:n
476         {
477             \tl_if_blank:nF {#1}
478             { \__hook_parse_dot_label:nN {#1} \c_false_bool }
479         }
480     }
481 }
482 \cs_new_protected:Npn \__hook_set_default_label:n #1
483 {
484     \str_if_eq:nnTF {#1} { top-level }
485     {
486         \msg_error:nnnnn { hooks } { set-top-level }
487         { to } { SetDefaultHookLabel } {#1}
488     }
489     { \tl_gset:Nn \g__hook_hook_curr_name_tl {#1} }
490 }

```

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

4.6 Adding or removing hook code

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

```

\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
\__hook_hash_check:nTF
\__hook_hash_check_aux:w
491 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_gput_code:nnn}
492 <latexrelease>{Hooks~with~args}
493 \cs_new_protected:Npn \hook_gput_code:nnn #1 #2 #3
494 {
495     \__hook_replacing_args_false:
496     \__hook_normalize_hook_args:Nnn \__hook_gput_code:nnn {#1} {#2} {#3}
497     \__hook_replacing_args_reset:
498 }
499 \cs_new_protected:Npn \hook_gput_code_with_args:nnn #1 #2 #3
500 {
501     \__hook_replacing_args_true:
502     \__hook_normalize_hook_args:Nnn \__hook_gput_code:nnn {#1} {#2} {#3}
503     \__hook_replacing_args_reset:
504 }

```

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:`.

```

505 \cs_new_protected:Npn \__hook_gput_code:nnn #1 #2 #3
506 {
507     \__hook_chk_args_allowed:nn {#1} { AddToHook }

```

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

```

508     \__hook_if_execute_immediately:nTF {#1}
509     {

```

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

```

510         \__hook_if_replacing_args:TF
511         {

```

```

512         \msg_error:nnnn { hooks } { one-time-args }
513         {#1} { AddToHook }
514     }
515     { }
516     \use:n
517 }
518 { \__hook_gput_code_store:nnn {#1} {#2} }
519     {#3}
520 }

521 \cs_new_protected:Npn \__hook_gput_code_store:nnn #1 #2 #3
522 {

```

Then check if the hook is usable.

```

523     \__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.

```

524     {
525         \__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.

```

526         \__hook_update_hook_code:n {#1}
527     }

```

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.

```

528     {
529         \__hook_if_disabled:nTF {#1}
530         { \msg_error:nnn { hooks } { hook-disabled } {#1} }
531         { \__hook_try_declaring_generic_hook:nnn {#1} {#2} {#3} }
532     }
533 }

```

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

```

534 \cs_new_protected:Npn \__hook_hook_gput_code_do:nnn #1 #2 #3
535 {

```

However, first some debugging info if debugging is enabled:

```

536     \__hook_debug:n{\iow_term:e{[lthooks]~ Add~ to~
537         \__hook_if_usable:nF {#1} { undeclared~ }
538         hook~ '#1'~ (#2) \on@line
539         ^^J [lthooks] \@spaces
540         <-- \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>`.

```

541     \str_if_eq:nnTF {#2} { top-level }
542     {
543         \str_if_eq:eeTF { top-level } { \__hook_currname_or_default: }
544         {

```

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

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

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

```
546         \__hook_cs_gput_right:nnn { _toplevel } {#1} {#3}
547     }
548     { \msg_error:nnn { hooks } { misused-top-level } {#1} }
549 }
550 {
```

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. We skip this step if there are no hashes at all in the argument: the token-by-token approach otherwise becomes a major performance issue when the contents of the hook are long.

```
551     \exp_args:Nne \__hook_prop_gput_labeled_cleanup:nnn
552     {
553         \__hook_if_replacing_args:TF
554         { \exp_not:n }
555         {
556             \__hook_hash_check:nTF {#3}
557             { \exp_not:n }
558             { \__hook_double_hashes:n }
559         }
560         {#3}
561     }
562     {#1} {#2}
563 }
564 }
565 \cs_new:Npe \__hook_hash_check:nTF #1
566 {
567     \exp_not:N \exp_after:wN \exp_not:N \__hook_hash_check_aux:w
568     \exp_not:N \tl_to_str:n {#1} \c_hash_str \c_hash_str
569     \exp_not:N \q_stop
570 }
571 \use:e
572 {
573     \cs_new:Npn \exp_not:N \__hook_hash_check_aux:w
574     #1 \c_hash_str #2 \c_hash_str #3 \exp_not:N \q_stop
575 }
576 { \tl_if_empty:nTF {#3} }
```

Adds code to a hook's code pool.

```
577 \cs_new_protected:Npn \__hook_prop_gput_labeled_cleanup:nnn #1 #2 #3
578 {
579     \tl_set:Nn \l__hook_return_tl {#1}
580     \__hook_if_replacing_args:TF
581     {
582         \__hook_if_usable:nT {#2}
583         {
584             \__hook_set_normalize_fn:nn {#2}
585             { Invalid-code-added-\msg_line_context: }
586             \__hook_normalize_fn:nn {#3} {#1}
587             \prop_get:NnN \l__hook_work_prop {#3} \l__hook_return_tl
```

```

588     }
589   }
590   { }
591   \exp_args:NcV \__hook_prop_gput_labeled_do:Nnn
592   { g__hook_#2_code_prop } \l__hook_return_tl {#3}
593 }
594 \cs_new_protected:Npn \__hook_prop_gput_labeled_do:Nnn #1 #2 #3
595 {
596   \prop_get:NnNTF #1 {#3} \l__hook_return_tl
597   { \prop_gput:Nno #1 {#3} { \l__hook_return_tl #2 } }
598   { \prop_gput:Nnn #1 {#3} {#2} }
599 }
600 \<latexrelease>\EndIncludeInRelease
601 \<latexrelease>\IncludeInRelease{2020/10/01}{\hook_gput_code:nnn}
602 \<latexrelease>      {Providing~hooks}
603 \<latexrelease>\cs_gset_protected:Npn \hook_gput_code:nnn #1 #2
604 \<latexrelease> { \__hook_normalize_hook_args:Nnn
605 \<latexrelease>      \__hook_gput_code:nnn {#1} {#2} }
606 \<latexrelease>\cs_gset_protected:Npn \__hook_gput_code:nnn #1 #2 #3
607 \<latexrelease> {
608 \<latexrelease>      \__hook_if_execute_immediately:nTF {#1}
609 \<latexrelease>      {#3}
610 \<latexrelease>      {
611 \<latexrelease>          \__hook_if_usable:nTF {#1}
612 \<latexrelease>          {
613 \<latexrelease>              \__hook_hook_gput_code_do:nnn {#1} {#2} {#3}
614 \<latexrelease>              \__hook_update_hook_code:n {#1}
615 \<latexrelease>          }
616 \<latexrelease>          {
617 \<latexrelease>              \__hook_if_disabled:nTF {#1}
618 \<latexrelease>              { \msg_error:nnn { hooks } { hook-disabled } {#1} }
619 \<latexrelease>              { \__hook_try_declaring_generic_hook:nnn
620 \<latexrelease>                  {#1} {#2} {#3} }
621 \<latexrelease>          }
622 \<latexrelease>      }
623 \<latexrelease> }
624 \<latexrelease>\cs_gset_protected:Npn \__hook_hook_gput_code_do:nnn #1 #2 #3
625 \<latexrelease> {
626 \<latexrelease>      \__hook_debug:n{\iow_term:x{****~ Add~ to~
627 \<latexrelease>          \__hook_if_usable:nF {#1} { undeclared~ }
628 \<latexrelease>          hook~ #1~ (#2)
629 \<latexrelease>          \on@line\space <~ \tl_to_str:n{#3}} }
630 \<latexrelease>      \str_if_eq:nnTF {#2} { top-level }
631 \<latexrelease>      {
632 \<latexrelease>          \str_if_eq:eeTF { top-level }
633 \<latexrelease>          { \__hook_currname_or_default: }
634 \<latexrelease>          {
635 \<latexrelease>              \__hook_init_structure:n {#1}
636 \<latexrelease>              \__hook_tl_gput_right:cn { \__hook_toplevel~#1 } {#3}
637 \<latexrelease>          }
638 \<latexrelease>          { \msg_error:nnn { hooks } { misused-top-level } {#1} }
639 \<latexrelease>      }
640 \<latexrelease> {

```

```

641 <latexrelease> \prop_get:cnNTF
642 <latexrelease> { g__hook_#1_code_prop } {#2} \l__hook_return_tl
643 <latexrelease> {
644 <latexrelease> \prop_gput:cno { g__hook_#1_code_prop } {#2}
645 <latexrelease> { \l__hook_return_tl #3 }
646 <latexrelease> }
647 <latexrelease> { \prop_gput:cn { g__hook_#1_code_prop } {#2} {#3} }
648 <latexrelease> }
649 <latexrelease> }
650 <latexrelease> \cs_gset_protected:Npn \hook_gput_code_with_args:nnn #1#2#3 { }
651 <latexrelease> \EndIncludeInRelease

```

(End of definition for `\hook_gput_code:nnn` and others. These functions are documented on page 17.)

`\__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.

```

652 <latexrelease> \IncludeInRelease{2023/06/01}{\__hook_chk_args_allowed:nn}
653 <latexrelease> {Hooks~with~args}
654 \cs_new_protected:Npn \__hook_chk_args_allowed:nn #1 #2
655 {
656 \__hook_if_replacing_args:TF
657 {
658 \__hook_if_declared:nT {#1}
659 { \tl_if_empty:cT { c__hook_#1_parameter_tl } { \use_ii:nn } }
660 \use_none:n
661 {
662 \msg_error:nnnn { hooks } { without-args } {#1} {#2}
663 \__hook_replacing_args_false:
664 }
665 }
666 { }
667 }
668 <latexrelease> \EndIncludeInRelease
669 <latexrelease> \IncludeInRelease{2020/10/01}{\__hook_chk_args_allowed:nn}
670 <latexrelease> {Hooks~with~args}
671 <latexrelease> \cs_undefine:N \__hook_chk_args_allowed:nn
672 <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 it is declared. An implicitly declared hook doesn't have arguments (in principle), so use `\c_false_bool` here.

```

673 \cs_new_protected:Npn \__hook_gput_undeclared_hook:nnn #1 #2 #3
674 {
675 \__hook_init_structure:n {#1}
676 \__hook_hook_gput_code_do:nnn {#1} {#2} {#3}
677 }

```

(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:nnnn` 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.

```

678 <latexrelease>\IncludeInRelease{2023/06/01}
679 <latexrelease>          {\_hook_try_declaring_generic_hook:nnn}
680 <latexrelease>          {Hooks-with-args}
681 \cs_new_protected:Npn \_hook_try_declaring_generic_hook:nnn #1
682   {
683     \_hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
684     \_hook_gput_code:nnn
685     \_hook_gput_undeclared_hook:nnn
686     {#1}
687   }
688 \cs_new_protected:Npn \_hook_try_declaring_generic_next_hook:nn #1
689   {
690     \_hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
691     \_hook_gput_next_code:nn
692     \_hook_gput_next_do:nn
693     {#1}
694   }
695 <latexrelease>\EndIncludeInRelease
696 <latexrelease>\IncludeInRelease{2021/11/15}
697 <latexrelease>          {\_hook_try_declaring_generic_hook:nnn}
698 <latexrelease>          {Standardize-generic-hook-names}
699 <latexrelease>\cs_gset_protected:Npn \_hook_try_declaring_generic_hook:nnn #1
700 <latexrelease>   {
701 <latexrelease>     \_hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop:
702 <latexrelease>       {#1}
703 <latexrelease>     \hook_gput_code:nnn
704 <latexrelease>     \_hook_gput_undeclared_hook:nnn
705 <latexrelease>     {#1}
706 <latexrelease>   }
707 <latexrelease>\cs_gset_protected:Npn
708 <latexrelease>   \_hook_try_declaring_generic_next_hook:nn #1
709 <latexrelease>   {
710 <latexrelease>     \_hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop:
711 <latexrelease>       {#1}
712 <latexrelease>     \hook_gput_next_code:nn
713 <latexrelease>     \_hook_gput_next_do:nn
714 <latexrelease>     {#1}
715 <latexrelease>   }
716 <latexrelease>\EndIncludeInRelease
717 <latexrelease>\IncludeInRelease{2020/10/01}
718 <latexrelease>          {\_hook_try_declaring_generic_hook:nnn}
719 <latexrelease>          {Standardize-generic-hook-names}
720 <latexrelease>\cs_new_protected:Npn

```

```

721 <latexrelease> \_hook_try_declaring_generic_hook:nnn #1
722 <latexrelease> {
723 <latexrelease> \_hook_try_declaring_generic_hook:nNNnn {#1}
724 <latexrelease> \hook_gput_code:nnn \_hook_gput_undeclared_hook:nnn
725 <latexrelease> }
726 <latexrelease> \cs_new_protected:Npn
727 <latexrelease> \_hook_try_declaring_generic_next_hook:nn #1
728 <latexrelease> {
729 <latexrelease> \_hook_try_declaring_generic_hook:nNNnn {#1}
730 <latexrelease> \hook_gput_next_code:nn \_hook_gput_next_do:nn
731 <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.

```

732 <latexrelease> \cs_new_protected:Npn \_hook_try_declaring_generic_hook:nNNnn #1
733 <latexrelease> {
734 <latexrelease> \_hook_if_file_hook:wTF #1 // \s__hook_mark
735 <latexrelease> {
736 <latexrelease> \exp_args:Ne
737 <latexrelease> \_hook_try_declaring_generic_hook_split:nNNnn
738 <latexrelease> { \exp_args:Ne \_hook_file_hook_normalize:n {#1} }
739 <latexrelease> }
740 <latexrelease> { \_hook_try_declaring_generic_hook_split:nNNnn {#1} }
741 <latexrelease> }
742 <latexrelease> \cs_new_protected:Npn
743 <latexrelease> \_hook_try_declaring_generic_hook_split:nNNnn #1 #2 #3
744 <latexrelease> {
745 <latexrelease> \_hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop:
746 <latexrelease> {#1}
747 <latexrelease> { #2 }
748 <latexrelease> { #3 } {#1}
749 <latexrelease> }
750 <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

```

751 <latexrelease> \IncludeInRelease{2023/06/01}
752 <latexrelease> {\_hook_try_declaring_generic_hook:wn}
753 <latexrelease> {Hooks-with-args}
754 \prg_new_protected_conditional:Npnn
755 \_hook_try_declaring_generic_hook:wn
756 #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
757 {
758 \_hook_if_generic:nTF {#5}
759 {
760 \_hook_if_usable:nF {#5}
761 {

```

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 `ltxcmdhooks`) will generate an appropriate error message.

```

762         \str_if_eq:nnT {#1} { cmd }
763         {
764             \__hook_try_put_cmd_hook:n {#5}
765             \__hook_make_usable:nn {#5} { 9 }
766             \use_none:nnn
767         }

```

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.

```

768         \__hook_make_usable:nn {#5} { 0 }
769     }
770     \__hook_if_generic_reversed:nT {#5}
771     { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
772     \prg_return_true:
773 }
774 {

```

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:.`

```

775         \__hook_if_deprecated_generic:nTF {#5}
776         {
777             \__hook_deprecated_generic_warn:n {#5}
778             \exp_after:wN \__hook_declare_deprecated_generic:NNn
779             \exp:w % \exp_end:
780         }
781         { \prg_return_false: }
782     }
783 }

```

`\__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).

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

```

784 \cs_new_protected:Npn \__hook_deprecated_generic_warn:n #1
785 { \__hook_deprecated_generic_warn:w #1 \s__hook_mark }
786 \cs_new_protected:Npn \__hook_deprecated_generic_warn:w
787 #1 / #2 / #3 \s__hook_mark
788 {
789     \if_cs_exist:w __hook~#1/#2/#3 \cs_end: \else:
790         \msg_warning:nnnnn { hooks } { generic-deprecated } {#1} {#2} {#3}
791     \fi:

```

```

792 \cs_gset_eq:cN { __hook~#1/#2/#3 } \scan_stop:
793 }

```

Now that the user has been told about the deprecation, we proceed by swapping $\langle name \rangle$ and $\langle place \rangle$ 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
794 \cs_new_protected:Npn \__hook_do_deprecated_generic:Nn #1 #2
795 { \__hook_do_deprecated_generic:Nw #1 #2 \s__hook_mark }
796 \cs_new_protected:Npn \__hook_do_deprecated_generic:Nw #1
797 #2 / #3 / #4 \s__hook_mark
798 { #1 { #2 / #4 / #3 } }
799 \cs_new_protected:Npn \__hook_declare_deprecated_generic:NNn #1 #2 #3
800 { \__hook_declare_deprecated_generic:NNw #1 #2 #3 \s__hook_mark }
801 \cs_new_protected:Npn \__hook_declare_deprecated_generic:NNw #1 #2
802 #3 / #4 / #5 \s__hook_mark
803 {
804 \__hook_try_declaring_generic_hook:wnTF #3 / #5 / #4 / \scan_stop:
805 { #3 / #5 / #4 }
806 #1 #2 { #3 / #5 / #4 }
807 }
808 \<latexrelease>\EndIncludeInRelease

809 \<latexrelease>\IncludeInRelease{2021/11/15}
810 \<latexrelease> {\__hook_try_declaring_generic_hook:wn}
811 \<latexrelease> {Standardize~generic~hook~names}
812 \<latexrelease>\prg_new_protected_conditional:Npnn
813 \<latexrelease> \__hook_try_declaring_generic_hook:wn
814 \<latexrelease> #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
815 \<latexrelease> {
816 \<latexrelease> \__hook_if_generic:nTF {#5}
817 \<latexrelease> {
818 \<latexrelease> \__hook_if_usable:nF {#5}
819 \<latexrelease> {
820 \<latexrelease> \str_if_eq:nnT {#1} { cmd }
821 \<latexrelease> { \__hook_try_put_cmd_hook:n {#5} }
822 \<latexrelease> \__hook_make_usable:n {#5}
823 \<latexrelease> }
824 \<latexrelease> \__hook_if_generic_reversed:nT {#5}
825 \<latexrelease> { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
826 \<latexrelease> \prg_return_true:
827 \<latexrelease> }
828 \<latexrelease> {
829 \<latexrelease> \__hook_if_deprecated_generic:nTF {#5}
830 \<latexrelease> {
831 \<latexrelease> \__hook_deprecated_generic_warn:n {#5}
832 \<latexrelease> \exp_after:wN \__hook_declare_deprecated_generic:NNn
833 \<latexrelease> \exp:w % \exp_end:
834 \<latexrelease> }
835 \<latexrelease> { \prg_return_false: }
836 \<latexrelease> }
837 \<latexrelease> }
838 \<latexrelease>\EndIncludeInRelease

839 \<latexrelease>\IncludeInRelease{2021/06/01}
840 \<latexrelease> {\__hook_try_declaring_generic_hook:wn}
841 \<latexrelease> {Support~cmd~hooks}

```

```

842 <latexrelease>\prg_new_protected_conditional:Npnn
843 <latexrelease>    \__hook_try_declaring_generic_hook:wn
844 <latexrelease>    #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
845 <latexrelease>    {
846 <latexrelease>        \tl_if_empty:nTF {#2}
847 <latexrelease>        { \prg_return_false: }
848 <latexrelease>        {
849 <latexrelease>            \prop_if_in:NnTF \c__hook_generics_prop {#1}
850 <latexrelease>            {
851 <latexrelease>                \__hook_if_usable:nF {#5}
852 <latexrelease>                {
853 <latexrelease>                    \str_if_eq:nnT {#1} { cmd }
854 <latexrelease>                    { \__hook_try_put_cmd_hook:n {#5} }
855 <latexrelease>                    \__hook_make_usable:n {#5}
856 <latexrelease>                }
857 <latexrelease>            \prop_if_in:NnTF
858 <latexrelease>            \c__hook_generics_reversed_ii_prop {#2}
859 <latexrelease>            { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
860 <latexrelease>            {
861 <latexrelease>                \prop_if_in:NnT
862 <latexrelease>                \c__hook_generics_reversed_iii_prop {#3}
863 <latexrelease>                { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
864 <latexrelease>            }
865 <latexrelease>            \prg_return_true:
866 <latexrelease>        }
867 <latexrelease>    { \prg_return_false: }
868 <latexrelease>    }
869 <latexrelease> }
870 <latexrelease>\EndIncludeInRelease

871 <latexrelease>\IncludeInRelease{2020/10/01}
872 <latexrelease>    {\__hook_try_declaring_generic_hook:wn}
873 <latexrelease>    {Support~cmd~hooks}
874 <latexrelease>\prg_new_protected_conditional:Npnn
875 <latexrelease>    \__hook_try_declaring_generic_hook:wn
876 <latexrelease>    #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
877 <latexrelease>    {
878 <latexrelease>        \tl_if_empty:nTF {#2}
879 <latexrelease>        { \prg_return_false: }
880 <latexrelease>        {
881 <latexrelease>            \prop_if_in:NnTF \c__hook_generics_prop {#1}
882 <latexrelease>            {
883 <latexrelease>                \__hook_if_declared:nF {#5} { \hook_new:n {#5} }
884 <latexrelease>            \prop_if_in:NnTF
885 <latexrelease>            \c__hook_generics_reversed_ii_prop {#2}
886 <latexrelease>            { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
887 <latexrelease>            {
888 <latexrelease>                \prop_if_in:NnT
889 <latexrelease>                \c__hook_generics_reversed_iii_prop {#3}
890 <latexrelease>                { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
891 <latexrelease>            }
892 <latexrelease>            \prg_return_true:
893 <latexrelease>        }
894 <latexrelease>    { \prg_return_false: }
895 <latexrelease>    }

```

```

896 <latexrelease> }
897 <latexrelease>\EndIncludeInRelease

```

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

`\__hook_if_file_hook_p:w` `\__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>`.

```

898 <latexrelease>\IncludeInRelease{2021/11/15}{\__hook_if_file_hook:w}
899 <latexrelease> {Standardize-generic-hook-names}
900 <latexrelease>\EndIncludeInRelease
901 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_if_file_hook:w}
902 <latexrelease> {Standardize-generic-hook-names}
903 <latexrelease>\prg_new_conditional:Npnn \__hook_if_file_hook:w
904 <latexrelease> #1 / #2 / #3 \s__hook_mark { TF }
905 <latexrelease> {
906 <latexrelease> \str_if_eq:nnTF {#1} { file }
907 <latexrelease> {
908 <latexrelease> \bool_lazy_or:nnTF
909 <latexrelease> { \tl_if_empty_p:n {#3} }
910 <latexrelease> { \str_if_eq_p:nn {#3} { / } }
911 <latexrelease> { \prg_return_false: }
912 <latexrelease> {
913 <latexrelease> \prop_if_in:NnTF \c__hook_generics_file_prop {#2}
914 <latexrelease> { \prg_return_true: }
915 <latexrelease> { \prg_return_false: }
916 <latexrelease> }
917 <latexrelease> }
918 <latexrelease> { \prg_return_false: }
919 <latexrelease> }
920 <latexrelease>\EndIncludeInRelease

```

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

```

\__hook_file_hook_normalize:n
\__hook_strip_double_slash:n 921 <latexrelease>\IncludeInRelease{2021/11/15}{\__hook_file_hook_normalize:n}
\__hook_strip_double_slash:w 922 <latexrelease> {Standardize-generic-hook-names}
923 <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}`.

```

924 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_file_hook_normalize:n}
925 <latexrelease> {Standardize-generic-hook-names}
926 <latexrelease>\cs_new:Npn \__hook_file_hook_normalize:n #1
927 <latexrelease> { \__hook_strip_double_slash:n {#1} }
928 <latexrelease>\cs_new:Npn \__hook_strip_double_slash:n #1
929 <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.

```

930 <latexrelease>\cs_new:Npn \__hook_strip_double_slash:w #1/#2/#3//#4\s__hook_mark
931 <latexrelease> {
932 <latexrelease> \tl_if_empty:nTF {#4}
933 <latexrelease> { #1/#2/#3 }
934 <latexrelease> { \__hook_strip_double_slash:w #1/#2/#3 /#4\s__hook_mark }
935 <latexrelease> }
936 <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

```

937 <latexrelease>\IncludeInRelease{2021/11/15}{\c__hook_generics_prop}
938 <latexrelease> {Standardize~generic~hook~names}
939 \clist_map_inline:nn { cmd , env , file , package , class , include }
940 {
941 \tl_const:cn { c__hook_generic_#1/.before_tl } { + }
942 \tl_const:cn { c__hook_generic_#1/.after_tl } { - }
943 }
944 \tl_const:cn { c__hook_generic_env/.begin_tl } { + }
945 \tl_const:cn { c__hook_generic_env/.end_tl } { + }
946 \tl_const:cn { c__hook_generic_include/.end_tl } { - }
947 \tl_const:cn { c__hook_generic_include/.excluded_tl } { + }

```

Deprecated generic hooks:

```

948 \clist_map_inline:nn { file , package , class , include }
949 {
950 \tl_const:cn { c__hook_deprecated_#1/.before_tl } { }
951 \tl_const:cn { c__hook_deprecated_#1/.after_tl } { }
952 }
953 \tl_const:cn { c__hook_deprecated_include/.end_tl } { }
954 <latexrelease>\EndIncludeInRelease
955 <latexrelease>\IncludeInRelease{2020/10/01}{\c__hook_generics_prop}
956 <latexrelease> {Standardize~generic~hook~names}
957 <latexrelease>\prop_const_from_keyval:Nn \c__hook_generics_prop
958 <latexrelease> {cmd=,env=,file=,package=,class=,include=}
959 <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):

```

960 <latexrelease>\IncludeInRelease{2021/11/15}{\c__hook_generics_reversed_ii_prop}
961 <latexrelease> {Standardize~generic~hook~names}
962 <latexrelease>\EndIncludeInRelease

```

```

963 <latexrelease>\IncludeInRelease{2020/10/01}{\c__hook_generics_reversed_ii_prop}
964 <latexrelease>{Standardize-generic-hook-names}
965 <latexrelease>\prop_const_from_keyval:Nn
966 <latexrelease>\c__hook_generics_reversed_ii_prop {after=,end=}
967 <latexrelease>\prop_const_from_keyval:Nn
968 <latexrelease>\c__hook_generics_reversed_iii_prop {after=}
969 <latexrelease>\prop_const_from_keyval:Nn
970 <latexrelease>\c__hook_generics_file_prop {before=,after=}
971 <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.

```

972 <latexrelease>\IncludeInRelease{2023/06/01}{\c__hook_parameter_cmd/.before_tl}
973 <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.

```

974 \tl_const:cn { c__hook_parameter_cmd/.before_tl } { #1#2#3#4#5#6#7#8#9 }
975 \tl_const:cn { c__hook_parameter_cmd/.after_tl } { #1#2#3#4#5#6#7#8#9 }
976 <latexrelease>\EndIncludeInRelease
977 <latexrelease>\IncludeInRelease{2020/10/01}{\c__hook_parameter_cmd/.before_tl}
978 <latexrelease>{Hooks-with-args}
979 <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.

```

980 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_gremove_code:nn}
981 <latexrelease>{Hooks-with-args}
982 \cs_new_protected:Npn \hook_gremove_code:nn #1 #2
983 { \__hook_normalize_hook_args:Nnn \__hook_gremove_code:nn {#1} {#2} }
984 \cs_new_protected:Npn \__hook_gremove_code:nn #1 #2
985 {

```

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).

```

986 \__hook_if_structure_exist:nTF {#1}
987 {

```

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>.

```

988 \str_if_eq:nnTF {#2} {*}
989 {
990 \prop_gclear:c { g__hook_#1_code_prop }
991 \__hook_toplevel_gset:nn {#1} { }
992 \__hook_next_gset:nn {#1} { }

```

```

993     }
994     {

```

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

```

995         \str_if_eq:nnTF {#2} { top-level }
996         { \__hook_toplevel_gset:nn {#1} { } }
997         {
998             \prop_gpop:cnNF { g__hook_#1_code_prop }
999             {#2} \l__hook_return_tl
1000             { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
1001         }
1002     }

```

Finally update the code, if the hook exists.

```

1003         \__hook_if_usable:nT {#1}
1004         { \__hook_update_hook_code:n {#1} }
1005     }

```

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

```

1006     {
1007         \__hook_if_deprecated_generic:nTF {#1}
1008         {
1009             \__hook_deprecated_generic_warn:n {#1}
1010             \__hook_do_deprecated_generic:Nn
1011             \__hook_gremove_code:nn {#1} {#2}
1012         }
1013         { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
1014     }

```

```

1015 }
1016 <latexrelease>\EndIncludeInRelease

1017 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_gremove_code:nn}
1018 <latexrelease>{Hooks~with~args}
1019 <latexrelease>\cs_new_protected:Npn \__hook_gremove_code:nn #1 #2
1020 <latexrelease> {
1021 <latexrelease>     \__hook_if_structure_exist:nTF {#1}
1022 <latexrelease>     {
1023 <latexrelease>         \str_if_eq:nnTF {#2} {*}
1024 <latexrelease>         {
1025 <latexrelease>             \prop_gclear:c { g__hook_#1_code_prop }
1026 <latexrelease>             \__hook_tl_gclear:c { __hook_toplevel~#1 }
1027 <latexrelease>             \__hook_tl_gclear:c { __hook_next~#1 }
1028 <latexrelease>         }
1029 <latexrelease>         {
1030 <latexrelease>             \str_if_eq:nnTF {#2} { top-level }
1031 <latexrelease>             { \__hook_tl_gclear:c { __hook_toplevel~#1 } }
1032 <latexrelease>             {
1033 <latexrelease>                 \prop_gpop:cnNF { g__hook_#1_code_prop }
1034 <latexrelease>                 {#2} \l__hook_return_tl
1035 <latexrelease>                 { \msg_warning:nnnn { hooks } { cannot-remove }
1036 <latexrelease>                     {#1} {#2} }
1037 <latexrelease>             }
1038 <latexrelease>         }
1039 <latexrelease>         \__hook_if_usable:nT {#1}
1040 <latexrelease>         { \__hook_update_hook_code:n {#1} }

```

```

1041 <latexrelease>      }
1042 <latexrelease>      {
1043 <latexrelease>          \__hook_if_deprecated_generic:nTF {#1}
1044 <latexrelease>          {
1045 <latexrelease>              \__hook_deprecated_generic_warn:n {#1}
1046 <latexrelease>              \__hook_do_deprecated_generic:Nn
1047 <latexrelease>              \__hook_gremove_code:nn {#1} {#2}
1048 <latexrelease>          }
1049 <latexrelease>          { \msg_warning:nnnn { hooks } { cannot-remove }
1050 <latexrelease>              {#1} {#2} }
1051 <latexrelease>      }
1052 <latexrelease>  }
1053 <latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_gremove_code:nn` and `\__hook_gremove_code:nn`. This function is documented on page 17.)

```

\__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, treating them correctly depending on their number of arguments, and depending on whether the code being added should have parameter tokens understood as parameters, or doubled to be stored as parameter tokens.

```

1054 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_cs_gput_right:nnn}
1055 <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.

```

1056 \cs_new_protected:Npn \__hook_cs_gput_right:nnn #1 #2
1057 {
1058   \if:w T
1059     \__hook_if_declared:nF {#2} { F }
1060     \tl_if_empty:cF { c__hook_#2_parameter_tl } { F }
1061     T
1062     \exp_after:wN \__hook_cs_gput_right_fast:nnn
1063   \else:
1064     \exp_after:wN \__hook_cs_gput_right_slow:nnn
1065   \fi:
1066   {#1} {#2}
1067 }
1068 \cs_new_protected:Npn \__hook_cs_gput_right_fast:nnn #1 #2 #3
1069 { \cs_gset:cpe { __hook#1~#2 }
1070   { \exp_not:v { __hook#1~#2 } \exp_not:n {#3} } }
1071 \cs_new_protected:Npn \__hook_cs_gput_right_slow:nnn #1 #2 #3
1072 {

```

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.

```

1073   \cs_if_exist:cF { __hook#1~#2 }
1074   { \__hook_code_gset_aux:nnn {#1} {#2} { } }
1075   \__hook_code_gset_auxi:eeen
1076   {
1077     \__hook_if_declared:nTF {#2}
1078     { \tl_use:c { c__hook_#2_parameter_tl } }
1079     {
1080       \__hook_if_generic:nTF {#2}

```

```

1081         { \__hook_generic_parameter:n {#2} }
1082         { \c__hook_nine_parameters_tl }
1083     }
1084 }

```

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 optimized by quickly checking `\cs_replacement_spec`.

```

1085 {
1086     \exp_args:NNo \exp_args:No \__hook_double_hashes:n
1087     {
1088         \cs:w __hook#1~#2 \exp_last_unbraced:Ne \cs_end:
1089         { \__hook_braced_cs_parameter:n { __hook#1~#2 } }
1090     }
1091 }

```

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

```

1092 {
1093     \__hook_if_replacing_args:TF
1094     { \exp_not:n }
1095     { \__hook_double_hashes:n }
1096     {#3}
1097 }

```

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

```

1098 { __hook#1~#2 }
1099 }

```

And as promised, the auxiliary that does the definition.

```

1100 \cs_new_protected:Npn \__hook_code_gset_auxi:nnnn #1 #2 #3 #4
1101 { \cs_gset:cpn {#4} #1 { #2 #3 } }
1102 \cs_generate_variant:Nn \__hook_code_gset_auxi:nnnn { eenen }
1103 <latexrelease>\EndIncludeInRelease
1104 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_cs_gput_right:nnn}
1105 <latexrelease> {Hooks~with~args}
1106 <latexrelease>\cs_undefine:N \__hook_cs_gput_right:nnn
1107 <latexrelease>\cs_undefine:N \__hook_cs_gput_right_fast:nnn
1108 <latexrelease>\cs_undefine:N \__hook_cs_gput_right_slow:nnn
1109 <latexrelease>\cs_undefine:N \__hook_code_gset_auxi:nnnn
1110 <latexrelease>\EndIncludeInRelease

```

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

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).

```

\__hook_code_gset:nn
\__hook_code_gset:ne
\__hook_toplevel_gset:nn
\__hook_next_gset:nn
\__hook_code_gset_aux:nnn
1111 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_code_gset:nn}
1112 <latexrelease> {Hooks~with~args}
1113 \cs_new_protected:Npn \__hook_code_gset:nn
1114 { \__hook_code_gset_aux:nnn { } }
1115 \cs_new_protected:Npn \__hook_toplevel_gset:nn
1116 { \__hook_code_gset_aux:nnn { _toplevel } }
1117 \cs_new_protected:Npn \__hook_next_gset:nn
1118 { \__hook_code_gset_aux:nnn { _next } }
1119 \cs_new_protected:Npn \__hook_code_gset_aux:nnn #1 #2 #3

```

```

1120 {
1121   \cs_gset:cpn { __hook#1~#2 \exp_last_unbraced:Ne }
1122   { \__hook_parameter:n {#2} }
1123   {#3}
1124 }
1125 \cs_generate_variant:Nn \__hook_code_gset:nn { ne }

1126 <latexrelease>\EndIncludeInRelease
1127 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_code_gset:nn}
1128 <latexrelease>{Hooks~with~args}
1129 <latexrelease>\cs_undefine:N \__hook_code_gset:nn
1130 <latexrelease>\cs_undefine:N \__hook_toplevel_gset:nn
1131 <latexrelease>\cs_undefine:N \__hook_next_gset:nn
1132 <latexrelease>\cs_undefine:N \__hook_code_gset_aux:nnn
1133 <latexrelease>\EndIncludeInRelease

```

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

__hook_normalize_cs_args:nn This macro normalizes 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.

```

1134 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_normalize_cs_args:nn}
1135 <latexrelease>{Hooks~with~args}
1136 \cs_new_protected:Npn \__hook_normalize_cs_args:nn #1 #2
1137 {
1138   \cs_if_exist:cT { __hook#1~#2 }
1139   {
1140     \__hook_code_gset_auxi:eeen
1141     { \tl_use:c { c__hook_#2_parameter_tl } }
1142     {
1143       \exp_args:NNo \exp_args:No \__hook_double_hashes:n
1144       {
1145         \cs:w __hook#1~#2 \exp_last_unbraced:Ne \cs_end:
1146         { \__hook_braced_cs_parameter:n { __hook#1~#2 } }
1147       }
1148     }
1149     { }
1150     { __hook#1~#2 }
1151   }
1152 }
1153 <latexrelease>\EndIncludeInRelease

1154 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_normalize_cs_args:nn}
1155 <latexrelease>{Hooks~with~args}
1156 <latexrelease>\cs_undefine:N \__hook_normalize_cs_args:nn
1157 <latexrelease>\EndIncludeInRelease

```

(End of definition for __hook_normalize_cs_args:nn.)

__hook_normalize_code_pool:n 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 T_EX 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.

```

1158 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_normalize_code_pool:n}
1159 <latexrelease>{Hooks~with~args}
1160 \cs_new_protected:Npn \__hook_normalize_code_pool:n #1
1161 {

```

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

```

1162   \__hook_set_normalize_fn:nn {#1} { Offending~label:~'##1' }
1163   \prop_clear:N \l__hook_work_prop
1164   \prop_map_function:cN { g__hook_#1_code_prop } \__hook_normalize_fn:nn
1165   \prop_gset_eq:cN { g__hook_#1_code_prop } \l__hook_work_prop
1166 }

```

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

```

1167 \cs_new_protected:Npn \__hook_set_normalize_fn:nn #1 #2
1168 {

```

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}

1169   \cs_set:Npn \__hook_tmp:w ##1##2##3##4##5##6##7##8##9 { }
1170   \tl_set:Ne \l__hook_tmpb_tl
1171   { \__hook_braced_cs_parameter:n { __hook_tmp:w } }
1172   \group_begin:
1173   \__hook_tl_set:cn { c__hook_hash_tl } { \exp_not:N \c__hook_hashes_tl }
1174   \use:e
1175   {
1176   \group_end:
1177   \tl_set:Nn \exp_not:N \l__hook_tmpb_tl { \l__hook_tmpb_tl }
1178   }

```

And `\l__hook_tpa_tl` contains:

```

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

```

with $\langle n \rangle$ being the number of arguments declared for the hook.

```

1179   \exp_last_unbraced:NNf
1180   \cs_set:Npn \__hook_tmp:w { \__hook_parameter:n {#1} } { }
1181   \tl_set:Ne \l__hook_tpa_tl
1182   { \__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`.

```

1183   \cs_gset_protected:Npe \__hook_normalize_fn:nn ##1 ##2
1184   {

```

Here we'll define two auxiliary macros: the first one throws an error when it detects an invalid argument reference. It piggybacks on T_EX'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.

```
1185 \group_begin:
1186 \int_set:Nn \tex_escapechar:D { -1 }
1187 \cs_set:cpn
1188 {
1189     hook~'#1'. ^^J
1190     (hooks) \prg_replicate:nn { 13 } { ~ }
1191     #2 % more message text
1192 }
1193 \exp_not:v { c__hook_#1_parameter_tl }
1194 {##2}
1195 \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 $\langle n \rangle$ 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.

```
1196 \cs_set:Npn \exp_not:N \__hook_tmp:w
1197 \exp_not:V \c__hook_nine_parameters_tl
1198 {
1199     \prop_put:Nne \exp_not:N \l__hook_work_prop
1200     {##1} { \exp_not:N \__hook_double_hashes:n {##2} }
1201 }
```

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 $\langle n \rangle$ 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.

```
1202 \exp_not:N \__hook_tmp:w
1203 \exp_not:V \l__hook_tmpa_tl
1204 \exp_args:No \exp_not:o
1205 { \exp_after:wN \__hook_tmp:w \l__hook_tmpb_tl }
1206 }
1207 }
```

```

1208 \cs_new_eq:NN \__hook_normalize_fn:nn ?
1209 \<latexrelease>\EndIncludeInRelease

1210 \<latexrelease>\IncludeInRelease{2020/10/01}{\__hook_normalize_code_pool:n}
1211 \<latexrelease>{Hooks~with~args}
1212 \<latexrelease>\cs_undefine:N \__hook_normalize_code_pool:n
1213 \<latexrelease>\EndIncludeInRelease

```

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

```

\__hook_cs_if_empty_p:c
\__hook_cs_if_empty:cTF

```

```

1214 \<latexrelease>\IncludeInRelease{2023/06/01}{\__hook_cs_if_empty:c}
1215 \<latexrelease>{Hooks~with~args}
1216 \prg_new_conditional:Npn \__hook_cs_if_empty:c #1 { p, T, F, TF }
1217 {
1218   \if:w \scan_stop: \__hook_replacement_spec:c {#1} \scan_stop:
1219   \prg_return_true:
1220   \else:
1221   \prg_return_false:
1222   \fi:
1223 }
1224 \cs_new:Npn \__hook_replacement_spec:c #1
1225 {
1226   \exp_args:Nc \token_if_macro:NT {#1}
1227   { \cs_replacement_spec:c {#1} }
1228 }
1229 \<latexrelease>\EndIncludeInRelease

1230 \<latexrelease>\IncludeInRelease{2020/10/01}{\__hook_cs_if_empty:c}
1231 \<latexrelease>{Hooks~with~args}
1232 \<latexrelease>\cs_undefine:N \__hook_cs_if_empty:c
1233 \<latexrelease>\EndIncludeInRelease

```

(End of definition for __hook_normalize_code_pool:n, __hook_set_normalize_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.

```

1234 \<latexrelease>\IncludeInRelease{2023/06/01}{\__hook_braced_cs_parameter:n}
1235 \<latexrelease>{Hooks~with~args}
1236 \cs_new:Npn \__hook_braced_cs_parameter:n #1
1237 {
1238   \exp_last_unbraced:Ne \__hook_braced_hidden_loop:w
1239   { \exp_args:Nc \__hook_cs_parameter_count:N {#1} } ? \s__hook_mark
1240 }
1241 \cs_new:Npn \__hook_braced_hidden_loop:w #1
1242 {
1243   \if:w ? #1
1244   \__hook_use_i_delimit_by_s_mark:nw
1245   \fi:
1246   { \exp_not:N \c__hook_hash_tl #1 }
1247   \__hook_braced_hidden_loop:w
1248 }

```

```

1249 \cs_new:Npn \__hook_cs_parameter_count:N #1
1250 {
1251   \exp_last_unbraced:Nf \__hook_cs_parameter_count:w
1252   { \token_if_macro:NT #1 { \cs_parameter_spec:N #1 } }
1253   ? \__hook_cs_end:w ? \__hook_cs_end:w ? \__hook_cs_end:w
1254   ? \__hook_cs_end:w ? \__hook_cs_end:w ? \__hook_cs_end:w
1255   ? \__hook_cs_end:w ? \__hook_cs_end:w ? \__hook_cs_end:w
1256   \s__hook_mark
1257 }
1258 \cs_new:Npn \__hook_cs_parameter_count:w #1#2 #3#4 #5#6 #7#8
1259 { #2 #4 #6 #8 \__hook_cs_parameter_count:w }
1260 \cs_new:Npn \__hook_cs_end:w #1 \s__hook_mark { }
1261 \<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.

```

1262 \<latexrelease>\IncludeInRelease{2020/10/01}{\__hook_braced_cs_parameter:n}
1263 \<latexrelease>{Hooks~with~args}
1264 \<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.

```

1265 \<latexrelease>\IncludeInRelease{2023/06/01}{\__hook_braced_parameter:n}
1266 \<latexrelease>{Hooks~with~args}
1267 \cs_new:Npn \__hook_braced_parameter:n #1
1268 {
1269   \if_case:w
1270     \int_eval:n
1271     { \exp_args:Nv \str_count:n { c__hook_#1_parameter_tl } / 3 }
1272   \exp_stop_f:
1273   \or: {##1}
1274   \or: {##1} {##2}
1275   \or: {##1} {##2} {##3}
1276   \or: {##1} {##2} {##3} {##4}
1277   \or: {##1} {##2} {##3} {##4} {##5}
1278   \or: {##1} {##2} {##3} {##4} {##5} {##6}
1279   \or: {##1} {##2} {##3} {##4} {##5} {##6} {##7}
1280   \or: {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8}
1281   \or: {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8} {##9}
1282   \else:
1283     \msg_expandable_error:nnn { latex2e } { should-not-happen }
1284     { Invalid~parameter~spec. }
1285   \fi:
1286 }
1287 \<latexrelease>\EndIncludeInRelease
1288 \<latexrelease>\IncludeInRelease{2020/10/01}{\__hook_braced_parameter:n}
1289 \<latexrelease>{Hooks~with~args}
1290 \<latexrelease>\cs_undefine:N \__hook_braced_parameter:n
1291 \<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 $\langle parameter\ text \rangle$ of the hook.

```

1292 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_parameter:n}
1293 <latexrelease>{Hooks~with~args}
1294 \cs_new:Npn \__hook_parameter:n #1
1295 {
1296   \cs:w c__hook_
1297   \tl_if_exist:cTF { c__hook_#1_parameter_tl }
1298     { #1_parameter } { empty }
1299   _tl \cs_end:
1300 }
1301 \cs_new:Npn \__hook_generic_parameter:n #1
1302 { \__hook_generic_parameter:w #1 / / / \s__hook_mark }
1303 \cs_new:Npn \__hook_generic_parameter:w #1 / #2 / #3 / #4 \s__hook_mark
1304 {
1305   \cs_if_exist_use:cF { c__hook_parameter_#1/./#3_tl }
1306     { \c__hook_empty_tl }
1307 }
1308 <latexrelease>\EndIncludeInRelease
1309 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_parameter:n}
1310 <latexrelease>{Hooks~with~args}
1311 <latexrelease>\cs_undefine:N \__hook_parameter:n
1312 <latexrelease>\cs_undefine:N \__hook_generic_parameter:n
1313 <latexrelease>\EndIncludeInRelease

```

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

4.7 Setting rules for hooks code

`\g__hook_??_code_prop` Initially these variables simply used an empty “label” name (not two question marks).
`\__hook-??` This was a bit unfortunate, because then l3doc complains about `__` in the middle of a
`\g__hook_??_reversed_tl` command name when trying to typeset the documentation. However using a “normal”
`\c__hook_??_parameter_tl` 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.

```

1314 \prop_new:c { g__hook_??_code_prop }
1315 \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.

```

1316 \tl_new:c { g__hook_??_reversed_tl }

```

The parameter text for the “default” hook is empty.

```

1317 <latexrelease>\IncludeInRelease{2023/06/01}{\c__hook_??_parameter_tl}
1318 <latexrelease>{Hooks~with~args}
1319 \tl_const:cn { c__hook_??_parameter_tl } { }
1320 <latexrelease>\EndIncludeInRelease
1321 <latexrelease>\IncludeInRelease{2020/10/01}{\c__hook_??_parameter_tl}
1322 <latexrelease>{Hooks~with~args}
1323 <latexrelease>\cs_undefine:c { c__hook_??_parameter_tl }
1324 <latexrelease>\EndIncludeInRelease

```

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

`\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).

```

1325 \cs_new_protected:Npn \hook_gset_rule:nnnn #1#2#3#4
1326 {
1327   \__hook_normalize_hook_rule_args:Nnnnn \__hook_gset_rule:nnnn
1328   {#1} {#2} {#3} {#4}
1329 }
1330 \<latexrelease>\IncludeInRelease{2022/06/01}{\__hook_gset_rule:nnnn}
1331 \<latexrelease> {Refuse~setting~rule~for~one~time~hooks}
1332 \cs_new_protected:Npn \__hook_gset_rule:nnnn #1#2#3#4
1333 {
1334   \__hook_if_deprecated_generic:nT {#1}
1335   {
1336     \__hook_deprecated_generic_warn:n {#1}
1337     \__hook_do_deprecated_generic:Nn \__hook_gset_rule:nnnn {#1}
1338     {#2} {#3} {#4}
1339     \__hook_use_none_delimit_by_s_mark:w
1340   }
1341   \__hook_if_execute_immediately:nT {#1}
1342   {
1343     \msg_error:nnnnnn { hooks } { rule-too-late }
1344     {#1} {#2} {#3} {#4}
1345     \__hook_use_none_delimit_by_s_mark:w
1346   }

```

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

```

1347   \__hook_init_structure:n {#1}

```

Then we clear any previous relationship between both labels.

```

1348   \__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.

```

1349   \cs_if_exist_use:cTF { __hook_rule_#3_gset:nnn }
1350   {
1351     {#1} {#2} {#4}
1352     \__hook_update_hook_code:n {#1}
1353   }
1354   {
1355     \msg_error:nnnnnn { hooks } { unknown-rule }
1356     {#1} {#2} {#3} {#4}
1357   }
1358   \s__hook_mark
1359 }

```

```

1360 \<latexrelease>\EndIncludeInRelease
1361 \<latexrelease>\IncludeInRelease{2020/10/01}{\__hook_gset_rule:nnnn}
1362 \<latexrelease> {Refuse~setting~rule~for~one~time~hooks}
1363 \<latexrelease>\cs_new_protected:Npn \__hook_gset_rule:nnnn #1#2#3#4
1364 \<latexrelease> {
1365 \<latexrelease>   \__hook_if_deprecated_generic:nT {#1}

```

```

1366 \<latexrelease> {
1367 \<latexrelease> \_hook_deprecated_generic_warn:n {#1}
1368 \<latexrelease> \_hook_do_deprecated_generic:Nn \_hook_gset_rule:nnnn
1369 \<latexrelease> {#1} {#2} {#3} {#4}
1370 \<latexrelease> \exp_after:wN \use_none:nnnnnnnnn \use_none:n
1371 \<latexrelease> }
1372 \<latexrelease> \_hook_init_structure:n {#1}
1373 \<latexrelease> \_hook_rule_gclear:nnn {#1} {#2} {#4}
1374 \<latexrelease> \cs_if_exist_use:cTF { \_hook_rule_#3_gset:nnn }
1375 \<latexrelease> {
1376 \<latexrelease> {#1} {#2} {#4}
1377 \<latexrelease> \_hook_update_hook_code:n {#1}
1378 \<latexrelease> }
1379 \<latexrelease> {
1380 \<latexrelease> \msg_error:nnnnnn { hooks } { unknown-rule }
1381 \<latexrelease> {#1} {#2} {#3} {#4}
1382 \<latexrelease> }
1383 \<latexrelease> }
1384 \<latexrelease> \EndIncludeInRelease

```

(End of definition for `\hook_gset_rule:nnnn` and `\_hook_gset_rule:nnnn`. This function is documented on page 18.)

```

\_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.

```

1385 \cs_new_protected:Npn \_hook_rule_before_gset:nnn #1#2#3
1386 {
1387 \_hook_tl_gset:ce
1388 { g\_hook\_#1\_rule\_ \_hook_label\_pair:nn {#2} {#3} \_tl }
1389 { \_hook_label\_ordered:nnTF {#2} {#3} { < } { > } }
1390 }
1391 \cs_new_eq:cN { \_hook_rule_<_gset:nnn } \_hook_rule_before_gset:nnn
1392 \cs_new_protected:Npn \_hook_rule_after_gset:nnn #1#2#3
1393 {
1394 \_hook_tl_gset:ce
1395 { g\_hook\_#1\_rule\_ \_hook_label\_pair:nn {#3} {#2} \_tl }
1396 { \_hook_label\_ordered:nnTF {#3} {#2} { < } { > } }
1397 }
1398 \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.

```

1399 \cs_new_protected:Npn \_hook_rule_voids_gset:nnn #1#2#3
1400 {

```

```

1401 \__hook_tl_gset:ce
1402 { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
1403 { \__hook_label_ordered:nnTF {#2} {#3} { -> } { <- } }
1404 }

```

(End of definition for __hook_rule_voids_gset:nnn.)

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

```

1405 \cs_new_protected:cpn { \__hook_rule_incompatible-error_gset:nnn } #1#2#3
1406 { \__hook_tl_gset:cn
1407   { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
1408   { xE }
1409 }
1410 \cs_new_protected:cpn { \__hook_rule_incompatible-warning_gset:nnn } #1#2#3
1411 { \__hook_tl_gset:cn
1412   { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl }
1413   { xW }
1414 }

```

(End of definition for __hook_rule_incompatible-error_gset:nnn and __hook_rule_incompatible-warning_gset:nnn.)

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

```

1415 \cs_new_protected:Npn \__hook_rule_unrelated_gset:nnn #1#2#3 { }
1416 \cs_new_protected:Npn \__hook_rule_gclear:nnn #1#2#3
1417 { \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.

```

1418 \cs_new:Npn \__hook_label_pair:nn #1#2
1419 {
1420   \if_case:w \__hook_str_compare:nn {#1} {#2} \exp_stop_f:
1421     #1 | #1 % 0
1422   \or:   #1 | #2 % +1
1423   \else: #2 | #1 % -1
1424   \fi:
1425 }

```

(End of definition for __hook_label_pair:nn.)

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

```

1426 \prg_new_conditional:Npnn \__hook_label_ordered:nn #1#2 { TF }
1427 {
1428   \if_int_compare:w \__hook_str_compare:nn {#1} {#2} > 0 \exp_stop_f:
1429   \prg_return_true:
1430   \else:
1431   \prg_return_false:
1432   \fi:
1433 }

```

(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.

```

1434 \cs_new:Npn \__hook_if_label_case:nnnnn #1#2
1435 {
1436   \cs:w use_
1437   \if_case:w \__hook_str_compare:nn {#1} {#2}
1438     i \or: ii \else: iii \fi: :nnn
1439   \cs_end:
1440 }

```

(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.

```

1441 \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.

```

1442 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_initialize_all:}
1443 <latexrelease>{Hooks~with~args}
1444 \cs_new_protected:Npn \__hook_initialize_all:
1445 {

```

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.

```

1446   \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.

```

1447   \__hook_debug:n { \prop_gclear:N \g__hook_used_prop }
1448   \seq_map_inline:Nn \g__hook_all_seq
1449   {
1450     \tl_if_empty:cTF { c__hook_##1_parameter_tl }
1451     { \__hook_replacing_args_false: }
1452     { \__hook_replacing_args_true: }
1453     \__hook_update_hook_code:n {##1}
1454     \__hook_replacing_args_reset:
1455   }

```

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

```

1456   \__hook_debug:n
1457   {
1458     \iow_term:e { ^^J[lthooks]~ All~initialized~(non-empty)~hooks: }
1459     \prop_map_inline:Nn \g__hook_used_prop
1460     {
1461       \iow_term:e
1462       { ^^J ~ ##1 ~ -> ~ \cs_replacement_spec:c { __hook-##1 } ~ }
1463     }
1464   }

```

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

```

1465 \__hook_post_initialization_defs:
1466 }

1467 <latexrelease>\EndIncludeInRelease
1468 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_initialize_all:}
1469 <latexrelease> {Hooks~with~args}
1470 <latexrelease>\cs_gset_protected:Npn \__hook_initialize_all:
1471 <latexrelease> {
1472 <latexrelease> \cs_gset_eq:NN \__hook_update_hook_code:n
1473 <latexrelease> \__hook_initialize_hook_code:n
1474 <latexrelease> \__hook_debug:n { \prop_gc_clear:N \g__hook_used_prop }
1475 <latexrelease> \seq_map_inline:Nn \g__hook_all_seq
1476 <latexrelease> { \__hook_update_hook_code:n {##1} }
1477 <latexrelease> \__hook_debug:n
1478 <latexrelease> {
1479 <latexrelease> \iow_term:x{^^JAll~ initialized~ (non-empty)~ hooks:}
1480 <latexrelease> \prop_map_inline:Nn \g__hook_used_prop
1481 <latexrelease> {
1482 <latexrelease> \iow_term:x
1483 <latexrelease> { ^^J ~ ##1 ~ -> ~
1484 <latexrelease> \cs_replacement_spec:c { __hook~##1 } ~ }
1485 <latexrelease> }
1486 <latexrelease> }
1487 <latexrelease> \cs_gset_eq:NN \hook_use:n \__hook_use_initialized:n
1488 <latexrelease> \cs_gset_eq:NN \__hook_preamble_hook:n \use_none:n
1489 <latexrelease> }
1490 <@@=
1491 <latexrelease>\cs_gset_eq:NN \@expl@@@initialize@all@@
1492 <latexrelease> \__hook_initialize_all:
1493 <@@=hook>
1494 <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.

```

1495 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_initialize_hook_code:n}
1496 <latexrelease> {Hooks~with~args}
1497 \cs_new_protected:Npn \__hook_initialize_hook_code:n #1
1498 {
1499 \__hook_debug:n
1500 { \iow_term:e { ^^J[lthooks]~ 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.

```

1501 \__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.

```

1502 \__hook_if_usable:nT {#1}
1503 {
1504   \prop_if_empty:cTF { g__hook_#1_code_prop }
1505   {
1506     \__hook_code_gset:ne {#1}
1507     {

```

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.

```

1508       \exp_not:c { __hook_toplevel~#1 }
1509       \__hook_braced_parameter:n {#1}
1510       \exp_not:c { __hook_next~#1 }
1511       \__hook_braced_parameter:n {#1}
1512     }
1513   }
1514 {

```

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.

```

1515 \__hook_if_reversed:nTF {#1}
1516 { \cs_set_eq:NN \__hook_tl_gput:Nn \__hook_tl_gput_left:Nn
1517   \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_left:NV }
1518 { \cs_set_eq:NN \__hook_tl_gput:Nn \__hook_tl_gput_right:Nn
1519   \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.

```

1520 \prop_set_eq:Nc \l__hook_work_prop { g__hook_#1_code_prop }
1521 \__hook_initialize_single:ccn
1522 { __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.

```

1523 \__hook_debug:n
1524 { \exp_args:Nne \prop_gput:Nnn \g__hook_used_prop {#1} { } }
1525 }
1526 }
1527 }

```

```

1528 <latexrelease>\EndIncludeInRelease

```

```

1529 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_initialize_hook_code:n}
1530 <latexrelease> {Hooks~with~args}
1531 <latexrelease>\cs_gset_protected:Npn \__hook_initialize_hook_code:n #1
1532 <latexrelease> {
1533 <latexrelease>   \__hook_debug:n
1534 <latexrelease>   { \iow_term:x { ^^J Update~code~for~hook~'~#1'
1535 <latexrelease>     \on@line :^^J } }

```

```

1536 <latexrelease> \_hook_include_legacy_code_chunk:n {#1}
1537 <latexrelease> \_hook_if_usable:nT {#1}
1538 <latexrelease> {
1539 <latexrelease> \prop_if_empty:cTF { g\_hook\_#1\_code\_prop }
1540 <latexrelease> {
1541 <latexrelease> \_hook_tl_gset:co { \_hook~#1 }
1542 <latexrelease> {
1543 <latexrelease> \cs:w \_hook\_toplevel~#1 \exp\_after:wN \cs\_end:
1544 <latexrelease> \cs:w \_hook\_next~#1 \cs\_end:
1545 <latexrelease> }
1546 <latexrelease> }
1547 <latexrelease> {
1548 <latexrelease> \_hook_if_reversed:nTF {#1}
1549 <latexrelease> { \cs\_set\_eq:NN \_hook\_tl\_gput:Nn
1550 <latexrelease> \_hook\_tl\_gput\_left:Nn
1551 <latexrelease> \cs\_set\_eq:NN \_hook\_clist\_gput:NV
1552 <latexrelease> \clist\_gput\_left:NV }
1553 <latexrelease> { \cs\_set\_eq:NN \_hook\_tl\_gput:Nn
1554 <latexrelease> \_hook\_tl\_gput\_right:Nn
1555 <latexrelease> \cs\_set\_eq:NN \_hook\_clist\_gput:NV
1556 <latexrelease> \clist\_gput\_right:NV }
1557 <latexrelease> \prop\_set\_eq:Nc \l\_hook\_work\_prop
1558 <latexrelease> { g\_hook\_#1\_code\_prop }
1559 <latexrelease> \_hook\_initialize\_single:ccn
1560 <latexrelease> { \_hook~#1 } { g\_hook\_#1\_labels\_clist } {#1}
1561 <latexrelease> \_hook\_debug:n
1562 <latexrelease> { \exp\_args:NNx \prop\_gput:Nnn \g\_hook\_used\_prop
1563 <latexrelease> {#1} { } }
1564 <latexrelease> }
1565 <latexrelease> }
1566 <latexrelease> }
1567 <latexrelease> \EndIncludeInRelease

```

(End of definition for _hook_initialize_hook_code:n.)

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

_hook_seq_csname:n

FMi: note to myself: verify

```

1568 \cs\_new:Npn \_hook\_tl\_csname:n #1 { l\_hook\_label\_#1\_tl }
1569 \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 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:

\l_hook_labels_int
\l_hook_front_tl
\l_hook_rear_tl
\l_hook_label_0_tl

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

```

1570 \seq\_new:N \l\_hook\_labels\_seq

```

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

```

1571 \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 :

```

1572      \tl_new:N \l__hook_front_tl
1573      \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:

```

1574      \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 $\langle \text{hook-code-plist} \rangle$, $\langle \text{hook-code-tl} \rangle$, $\langle \text{hook-top-level-code-tl} \rangle$, $\langle \text{hook-next-code-tl} \rangle$, $\langle \text{hook-ordered-labels-clist} \rangle$ and $\langle \text{hook-name} \rangle$ (the latter is only used for debugging—the $\langle \text{hook-rule-plist} \rangle$ is accessed using the $\langle \text{hook-name} \rangle$).

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

```

1575 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_initialize_single:NNn}
1576 <latexrelease>{Hooks~with~args}
1577 \cs_new_protected:Npn \__hook_initialize_single:NNn #1#2#3
1578 {

```

Step T1: Initialize the data structure ...

```

1579      \seq_clear:N \l__hook_labels_seq
1580      \int_zero:N \l__hook_labels_int

```

Store the name of the hook:

```

1581      \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.

```

1582      \prop_map_inline:Nn \l__hook_work_prop
1583      {
1584          \int_incr:N \l__hook_labels_int
1585          \seq_put_right:Nn \l__hook_labels_seq {##1}
1586          \__hook_tl_set:cn { \__hook_tl_csname:n {##1} } { 0 }
1587          \seq_clear_new:c { \__hook_seq_csname:n {##1} }
1588      }

```

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.

```

1589   \prop_map_inline:Nn \l__hook_work_prop
1590   {
1591     \prop_map_inline:Nn \l__hook_work_prop
1592     {
1593       \__hook_if_label_case:nnnnn {##1} {####1}
1594       { \prop_map_break: }
1595       { \__hook_apply_label_pair:nnn {##1} {####1} }
1596       { \__hook_apply_label_pair:nnn {####1} {##1} }
1597       {#3}
1598     }
1599   }

```

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

```

1600   \__hook_debug:n { \__hook_debug_label_data:N \l__hook_work_prop }

```

Step T4:

```

1601   \tl_set:Nn \l__hook_rear_tl { 0 }
1602   \tl_set:cn { \__hook_tl_csname:n { 0 } } { 0 }
1603   \seq_map_inline:Nn \l__hook_labels_seq
1604   {
1605     \int_compare:nNnT { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
1606     {
1607       \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } } {##1}
1608       \tl_set:Nn \l__hook_rear_tl {##1}
1609     }
1610   }
1611   \tl_set_eq:Nc \l__hook_front_tl { \__hook_tl_csname:n { 0 } }
1612   \__hook_tl_gclear:N #1
1613   \clist_gclear:N #2

```

The whole loop gets combined in steps T5–T7:

```

1614   \bool_while_do:nn { ! \str_if_eq_p:Vn \l__hook_front_tl { 0 } }
1615   {

```

This part is step T5:

```

1616     \int_decr:N \l__hook_labels_int
1617     \prop_get:NVN \l__hook_work_prop \l__hook_front_tl \l__hook_return_tl
1618     \exp_args:NNV \__hook_tl_gput:Nn #1 \l__hook_return_tl

1619     \__hook_clist_gput:NV #2 \l__hook_front_tl
1620     \__hook_debug:n{ \iow_term:e[lthooks]~ 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.

```

1621     \seq_map_inline:cn { \__hook_seq_csname:n { \l__hook_front_tl } }

```

```

1622 {
1623   \tl_set:ce { \__hook_tl_csname:n {##1} }
1624   { \int_eval:n
1625     { \cs:w \__hook_tl_csname:n {##1} \cs_end: - 1 }
1626   }
1627   \int_compare:nNnT
1628     { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
1629   {
1630     \tl_set:cn
1631       { \__hook_tl_csname:n { \l__hook_rear_tl } } {##1}
1632     \tl_set:Nn \l__hook_rear_tl {##1}
1633   }
1634 }

```

and here is step T7:

```

1635   \tl_set_eq:Nc \l__hook_front_tl
1636   { \__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.

```

1637   }
1638   \int_compare:nNnF \l__hook_labels_int = 0
1639   {
1640     \iow_term:e{=====}
1641     \iow_term:e{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

```

1642   \__hook_debug_label_data:N \l__hook_work_prop
1643   \iow_term:e{=====}
1644 }

```

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.

```

1645   \exp_args:NNe \__hook_tl_gput:Nn #1
1646   { \exp_not:c { __hook_toplevel~#3 } \__hook_braced_parameter:n {#3} }
1647   \__hook_tl_gput_right:Ne #1
1648   { \exp_not:c { __hook_next~#3 } \__hook_braced_parameter:n {#3} }
1649   \use:e
1650   {
1651     \cs_gset:cpn { __hook~#3 } \use:c { c__hook_#3_parameter_tl }
1652     { \exp_not:V #1 }
1653   }
1654 }
1655 \cs_generate_variant:Nn \__hook_initialize_single:NNn { cc }
1656 \<latexrelease>\EndIncludeInRelease

```

```

1657 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_initialize_single:NNn}
1658 <latexrelease>      {Hooks~with~args}
1659 <latexrelease>\cs_new_protected:Npn \__hook_initialize_single:NNn #1#2#3
1660 <latexrelease>  {
1661 <latexrelease>    \seq_clear:N \l__hook_labels_seq
1662 <latexrelease>    \int_zero:N \l__hook_labels_int
1663 <latexrelease>    \tl_set:Nn \l__hook_cur_hook_tl {#3}
1664 <latexrelease>    \prop_map_inline:Nn \l__hook_work_prop
1665 <latexrelease>      {
1666 <latexrelease>        \int_incr:N \l__hook_labels_int
1667 <latexrelease>        \seq_put_right:Nn \l__hook_labels_seq {##1}
1668 <latexrelease>        \__hook_tl_set:cn { \__hook_tl_csname:n {##1} } { 0 }
1669 <latexrelease>        \seq_clear_new:c { \__hook_seq_csname:n {##1} }
1670 <latexrelease>      }
1671 <latexrelease>    \prop_map_inline:Nn \l__hook_work_prop
1672 <latexrelease>      {
1673 <latexrelease>        \prop_map_inline:Nn \l__hook_work_prop
1674 <latexrelease>          {
1675 <latexrelease>            \__hook_if_label_case:nnnnn {##1} {####1}
1676 <latexrelease>            { \prop_map_break: }
1677 <latexrelease>            { \__hook_apply_label_pair:nnn {##1} {####1} }
1678 <latexrelease>            { \__hook_apply_label_pair:nnn {####1} {##1} }
1679 <latexrelease>            {#3}
1680 <latexrelease>          }
1681 <latexrelease>        }
1682 <latexrelease>    \__hook_debug:n
1683 <latexrelease>      { \__hook_debug_label_data:N \l__hook_work_prop }
1684 <latexrelease>    \tl_set:Nn \l__hook_rear_tl { 0 }
1685 <latexrelease>    \tl_set:cn { \__hook_tl_csname:n { 0 } } { 0 }
1686 <latexrelease>    \seq_map_inline:Nn \l__hook_labels_seq
1687 <latexrelease>      {
1688 <latexrelease>        \int_compare:nNnT
1689 <latexrelease>          { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
1690 <latexrelease>          {
1691 <latexrelease>            \tl_set:cn { \__hook_tl_csname:n
1692 <latexrelease>              { \l__hook_rear_tl } } {##1}
1693 <latexrelease>            \tl_set:Nn \l__hook_rear_tl {##1}
1694 <latexrelease>          }
1695 <latexrelease>        }
1696 <latexrelease>    \tl_set_eq:Nc \l__hook_front_tl { \__hook_tl_csname:n { 0 } }
1697 <latexrelease>    \__hook_tl_gclear:N #1
1698 <latexrelease>    \clist_gclear:N #2
1699 <latexrelease>    \bool_while_do:nn
1700 <latexrelease>      { ! \str_if_eq_p:Vn \l__hook_front_tl { 0 } }
1701 <latexrelease>      {
1702 <latexrelease>        \int_decr:N \l__hook_labels_int
1703 <latexrelease>        \prop_get:NVN \l__hook_work_prop
1704 <latexrelease>          \l__hook_front_tl \l__hook_return_tl
1705 <latexrelease>        \exp_args:NNV \__hook_tl_gput:Nn #1 \l__hook_return_tl
1706 <latexrelease>        \__hook_clist_gput:NV #2 \l__hook_front_tl
1707 <latexrelease>        \__hook_debug:n{ \iow_term:x
1708 <latexrelease>          {Handled~ code~ for~ \l__hook_front_tl} }
1709 <latexrelease>        \seq_map_inline:cn
1710 <latexrelease>          { \__hook_seq_csname:n { \l__hook_front_tl } }

```

```

1711 <latexrelease> {
1712 <latexrelease> \tl_set:cx { \__hook_tl_csname:n {##1} }
1713 <latexrelease> { \int_eval:n
1714 <latexrelease> { \cs:w \__hook_tl_csname:n {##1} \cs_end: - 1 }
1715 <latexrelease> }
1716 <latexrelease> \int_compare:nNnT
1717 <latexrelease> { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
1718 <latexrelease> {
1719 <latexrelease> \tl_set:cn { \__hook_tl_csname:n
1720 <latexrelease> { \l__hook_rear_tl } } {##1}
1721 <latexrelease> \tl_set:Nn \l__hook_rear_tl {##1}
1722 <latexrelease> }
1723 <latexrelease> }
1724 <latexrelease> \tl_set_eq:Nc \l__hook_front_tl
1725 <latexrelease> { \__hook_tl_csname:n { \l__hook_front_tl } }
1726 <latexrelease> }
1727 <latexrelease> \int_compare:nNnF \l__hook_labels_int = 0
1728 <latexrelease> {
1729 <latexrelease> \iow_term:x{=====}
1730 <latexrelease> \iow_term:x{Error:~ label~ rules~ are~ incompatible:}
1731 <latexrelease> \__hook_debug_label_data:N \l__hook_work_prop
1732 <latexrelease> \iow_term:x{=====}
1733 <latexrelease> }
1734 <latexrelease> \exp_args:NNo \__hook_tl_gput:Nn #1
1735 <latexrelease> { \cs:w \__hook_toplevel~#3 \cs_end: }
1736 <latexrelease> \__hook_tl_gput_right:Nn #1 { \cs:w \__hook_next~#3 \cs_end: }
1737 <latexrelease> }
1738 <latexrelease> \cs_generate_variant:Nn \__hook_tl_gput_right:Nn { No }
1739 <latexrelease> \EndIncludeInRelease

```

(End of definition for __hook_initialize_single:NNn.)

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

```

1740 \cs_new:Npn \__hook_tl_gput:Nn { \ERROR }
1741 \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 This is the payload of steps T2 and T3 executed in the loop described above. This macro
__hook_label_if_exist_apply:nnnF 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.

```

1742 \cs_new_protected:Npn \__hook_apply_label_pair:nnn #1#2#3
1743 {

```

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.

```
1744 \__hook_label_if_exist_apply:nnnF {#1} {#2} {#3}
1745 {
```

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

```
1746 \__hook_label_if_exist_apply:nnnF {#1} {#2} { ?? } { }
1747 }
1748 }
```

```
1749 \cs_new_protected:Npn \__hook_label_if_exist_apply:nnnF #1#2#3
1750 {
1751 \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`.

```
1752 \__hook_apply_rule:nnn {#1} {#2} {#3}
1753 \exp_after:wN \use_none:n
1754 \else:
1755 \use:nn
1756 \fi:
1757 }
```

(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 $\langle label1 \rangle$, $\langle label2 \rangle$, $\langle hook-name \rangle$, and $\langle hook-code-plist \rangle$.

```
1758 \cs_new_protected:Npn \__hook_apply_rule:nnn #1#2#3
1759 {
1760 \cs:w __hook_apply_
1761 \cs:w g__hook_#3_reversed_tl \cs_end: rule_
1762 \cs:w g__hook_ #3 _rule_ #1 | #2 _tl \cs_end: :nnn \cs_end:
1763 {#1} {#2} {#3}
1764 }
```

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

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

```
\__hook_apply_rule_>:nnn
1765 \cs_new_protected:cpn { __hook_apply_rule_<:nnn } #1#2#3
1766 {
1767 \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
1768 \tl_set:ce { \__hook_tl_csname:n {#2} }
1769 { \int_eval:n{ \cs:w \__hook_tl_csname:n {#2} \cs_end: + 1 } }
1770 \seq_put_right:cn{ \__hook_seq_csname:n {#1} }{#2}
1771 }
1772 \cs_new_protected:cpn { __hook_apply_rule_>:nnn } #1#2#3
1773 {
1774 \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
1775 \tl_set:ce { \__hook_tl_csname:n {#1} }
1776 { \int_eval:n{ \cs:w \__hook_tl_csname:n {#1} \cs_end: + 1 } }
1777 \seq_put_right:cn{ \__hook_seq_csname:n {#2} }{#1}
1778 }
```

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

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

```

1779 \cs_new_protected:cpn { __hook_apply_rule_xE:nnn } #1#2#3
1780 {
1781   \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
1782   \msg_error:nnnnnn { hooks } { labels-incompatible }
1783   {#1} {#2} {#3} { 1 }
1784   \use:c { __hook_apply_rule_>:nnn } {#1} {#2} {#3}
1785   \use:c { __hook_apply_rule_<:nnn } {#1} {#2} {#3}
1786 }
1787 \cs_new_protected:cpn { __hook_apply_rule_xW:nnn } #1#2#3
1788 {
1789   \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
1790   \msg_warning:nnnnnn { hooks } { labels-incompatible }
1791   {#1} {#2} {#3} { 0 }
1792 }

```

(End of definition for `\__hook_apply_rule_xE:nnn` and `\__hook_apply_rule_xW: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
`\__hook_apply_rule_<:nnn` 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.¹⁰ 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.

```

1793 \cs_new_protected:cpn { __hook_apply_rule_>:nnn } #1#2#3
1794 {
1795   \__hook_debug:n
1796   {
1797     \__hook_msg_pair_found:nnn {#1} {#2} {#3}
1798     \iow_term:e{--->~ Drop~ '#2'~ code~ from~
1799       \iow_char:N \ g__hook_ \l__hook_cur_hook_tl _code_prop ~
1800       because~ of~ '#1' }
1801   }
1802   \prop_put:Nnn \l__hook_work_prop {#2} { }
1803 }
1804 \cs_new_protected:cpn { __hook_apply_rule_<:nnn } #1#2#3
1805 {
1806   \__hook_debug:n
1807   {
1808     \__hook_msg_pair_found:nnn {#1} {#2} {#3}
1809     \iow_term:e{--->~ Drop~ '#1'~ code~ from~
1810       \iow_char:N \ g__hook_ \l__hook_cur_hook_tl _code_prop ~
1811       because~ of~ '#2' }
1812   }
1813   \prop_put:Nnn \l__hook_work_prop {#1} { }
1814 }

```

¹⁰This 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   Reversed rules.
\__hook_apply_rule>:nnn   1815 \cs_new_eq:cc { __hook_apply_rule<:nnn } { __hook_apply_rule>:nnn }
\__hook_apply_rule<=:nnn  1816 \cs_new_eq:cc { __hook_apply_rule>:nnn } { __hook_apply_rule<=:nnn }
\__hook_apply_rule->:nnn  1817 \cs_new_eq:cc { __hook_apply_rule<=:nnn } { __hook_apply_rule->:nnn }
\__hook_apply_rule_xW:nnn 1818 \cs_new_eq:cc { __hook_apply_rule->:nnn } { __hook_apply_rule_xW:nnn }
\__hook_apply_rule_xE:nnn 1819 \cs_new_eq:cc { __hook_apply_rule_xE:nnn } { __hook_apply_rule_xE:nnn }
\__hook_apply_rule_xW:nnn 1820 \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.
                             1821 \cs_new_protected:Npn \__hook_msg_pair_found:nnn #1#2#3
                             1822 {
                             1823   \iow_term:e{~ \str_if_eq:nnTF {#3} {??} {default} {~normal} ~
                             1824     rule~ \__hook_label_pair:nn {#1} {#2}:~
                             1825     \use:c { g__hook_#3_rule_ \__hook_label_pair:nn {#1} {#2} _tl } ~
                             1826     found}
                             1827 }

```

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

```

\__hook_debug_label_data:N
                             1828 \cs_new_protected:Npn \__hook_debug_label_data:N #1 {
                             1829   \iow_term:e{Code~ labels~ for~ sorting:}
                             1830   \iow_term:e{~ \seq_use:Nnnn\l__hook_labels_seq {~and~}{,~}{~and~} }
                             1831   \iow_term:e{^^J Data~ structure~ for~ label~ rules:}
                             1832   \prop_map_inline:Nn #1
                             1833   {
                             1834     \iow_term:e{~ #1~ =~ \tl_use:c{ \__hook_tl_csname:n {##1} }~ ->~
                             1835     \seq_use:cnnn{ \__hook_seq_csname:n {##1} }{~>~}{~>~}{~>~}
                             1836   }
                             1837 }
                             1838 \iow_term:e{}
                             1839 }

```

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

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

```

\__hook_log_line:e
\__hook_log_line_indent:e
\__hook_log:nN
                             1840 \cs_new_protected:Npn \hook_log:n #1
                             1841 {
                             1842   \cs_set_eq:NN \__hook_log_cmd:e \iow_log:e
                             1843   \__hook_normalize_hook_args:Nn \__hook_log:nN {#1} \tl_log:e
                             1844 }
                             1845 \cs_if_exist:NTF \iow_show:e
                             1846 {
                             1847   \cs_new_protected:Npn \hook_show:n #1
                             1848   {
                             1849     \cs_set_eq:NN \__hook_log_cmd:e \iow_show:e
                             1850     \__hook_normalize_hook_args:Nn \__hook_log:nN {#1} \tl_show:e
                             1851   }

```

```

1852 }
1853 {
1854   \cs_new_protected:Npn \hook_show:n #1
1855   {
1856     \cs_set_eq:NN \__hook_log_cmd:e \iow_term:e
1857     \__hook_normalize_hook_args:Nn \__hook_log:nN {#1} \tl_show:e
1858   }
1859 }
1860 \cs_new_protected:Npn \__hook_log_line:e #1
1861 { \__hook_log_cmd:e { >~#1 } }
1862 \cs_new_protected:Npn \__hook_log_line_indent:e #1
1863 { \__hook_log_cmd:e { >~\@spaces #1 } }
1864 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_log:nN}
1865 <latexrelease>{Hooks~with~args}
1866 \cs_new_protected:Npn \__hook_log:nN #1 #2
1867 {
1868   \__hook_if_deprecated_generic:nT {#1}
1869   {
1870     \__hook_deprecated_generic_warn:n {#1}
1871     \__hook_do_deprecated_generic:Nn \__hook_log:nN {#1} #2
1872     \exp_after:wN \use_none:nnnnnnnn \use_none:nnnnn
1873   }
1874   \__hook_preamble_hook:n {#1}
1875   \__hook_log_cmd:e
1876   {
1877     ^^J ->~The~
1878     \__hook_if_generic:nT {#1} { generic~ }
1879     hook~'#1'
1880     \__hook_if_disabled:nF {#1}
1881     {
1882       \exp_args:Nne \__hook_print_args:nn {#1}
1883       {
1884         \int_eval:n
1885         { \str_count:e { \__hook_parameter:n {#1} } / 3 }
1886       }
1887     }
1888     :
1889   }
1890   \__hook_if_usable:nF {#1}
1891   { \__hook_log_line:e { The~hook~is~not~declared. } }
1892   \__hook_if_disabled:nT {#1}
1893   { \__hook_log_line:e { The~hook~is~disabled. } }
1894   \hook_if_empty:nTF {#1}
1895   { #2 { The~hook~is~empty } }
1896   {
1897     \__hook_log_line:e { Code~chunks: }
1898     \bool_lazy_or:nnTF
1899     { ! \prop_if_exist_p:c { g__hook_#1_code_prop } }
1900     { \prop_if_empty_p:c { g__hook_#1_code_prop } }
1901     { \__hook_log_line_indent:e { --- } }
1902     {
1903       \prop_map_inline:cn { g__hook_#1_code_prop }
1904       {

```

```

1905         \exp_after:wN \cs_set:Npn \exp_after:wN \__hook_tmp:w
1906         \c__hook_nine_parameters_tl {##2}
1907         \__hook_log_line_indent:e
1908         { ##1~-->~\cs_replacement_spec:N \__hook_tmp:w }
1909     }
1910 }

```

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

```

1911 \__hook_log_line:e
1912 {
1913     Document-level~(top-level)~code
1914     \__hook_if_usable:nT {#1}
1915     { ~(executed~\__hook_if_reversed:nTF {#1} {first} {last} ) } :
1916 }
1917 \__hook_log_line_indent:e
1918 {
1919     \__hook_cs_if_empty:cTF { __hook_toplevel~#1 }
1920     { --- }
1921     { -> ~ \cs_replacement_spec:c { __hook_toplevel~#1 } }
1922 }
1923 \__hook_log_line:e { Extra~code~for~next~invocation: }
1924 \__hook_log_line_indent:e
1925 {
1926     \__hook_cs_if_empty:cTF { __hook_next~#1 }
1927     { --- }

```

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.

```

1928     {
1929         -> ~ \exp_last_unbraced:Nf \__hook_log_next_code:w
1930         { \cs_replacement_spec:c { __hook_next~#1 } }
1931     }
1932 }

```

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.

```

1933 \__hook_log_line:e { Rules: }
1934 \bool_set_true:N \l__hook_tmpa_bool
1935 \__hook_list_rules:nn {#1}
1936 {
1937     \bool_set_false:N \l__hook_tmpa_bool
1938     \__hook_log_line_indent:e
1939     {
1940         ##2~ with~
1941         \str_if_eq:nnT {##3} {??} { default~ }
1942         relation~ ##1
1943     }
1944 }
1945 \bool_if:NT \l__hook_tmpa_bool
1946 { \__hook_log_line_indent:e { --- } }

```

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

```

1947 \bool_lazy_and:nnTF
1948 { \__hook_if_usable_p:n {#1} }
1949 { ! \hook_if_empty_p:n {#1} }
1950 {
1951   \__hook_log_line:e
1952   {
1953     Execution~order
1954     \bool_if:NTF \l__hook_tmpa_bool
1955     { \__hook_if_reversed:nT {#1} { ~(after~reversal) } }
1956     { ~(after~
1957       \__hook_if_reversed:nT {#1} { reversal~and~
1958       applying~rules)
1959     } :
1960   }
1961   #2 % \tl_show:n
1962   {
1963     \@spaces
1964     \clist_if_empty:cTF { g__hook_#1_labels_clist }
1965     { --- }
1966     { \clist_use:cn { g__hook_#1_labels_clist } { ,~ } }
1967   }
1968 }
1969 {
1970   \__hook_log_line:e { Execution~order: }
1971   #2
1972   {
1973     \@spaces Not~set~because~the~hook~ \__hook_if_usable:nTF {#1}
1974     { code~pool~is~empty }
1975     { is~\__hook_if_disabled:nTF {#1} {disabled} {undeclared} }
1976   }
1977 }
1978 }
1979 }
1980 \<latexrelease>\EndIncludeInRelease
1981 %
1982 \<latexrelease>\IncludeInRelease{2020/10/01}{\__hook_log:nN}
1983 \<latexrelease> {Hooks~with~args}
1984 \<latexrelease>\cs_new_protected:Npn \__hook_log:nN #1 #2
1985 \<latexrelease> {
1986   \<latexrelease> \__hook_if_deprecated_generic:nT {#1}
1987   \<latexrelease> {
1988     \<latexrelease> \__hook_deprecated_generic_warn:n {#1}
1989     \<latexrelease> \__hook_do_deprecated_generic:Nn \__hook_log:nN {#1} #2
1990     \<latexrelease> \exp_after:wN \use_none:nnnnnnnn \use_none:nnnnn
1991   }
1992   \<latexrelease> \__hook_preamble_hook:n {#1}
1993   \<latexrelease> \__hook_log_cmd:e
1994   \<latexrelease> { ^~J ->~The~ \__hook_if_generic:nT
1995     \<latexrelease> {#1} { generic~ } hook~'#1': }
1996   \<latexrelease> \__hook_if_usable:nF {#1}
1997   \<latexrelease> { \__hook_log_line:e { The~hook~is~not~declared. } }
1998   \<latexrelease> \__hook_if_disabled:nT {#1}
1999   \<latexrelease> { \__hook_log_line:e { The~hook~is~disabled. } }
2000   \<latexrelease> \hook_if_empty:nTF {#1}

```

```

2001 <latexrelease> { #2 { The-hook-is-empty } }
2002 <latexrelease> {
2003 <latexrelease> \_hook_log_line:e { Code-chunks: }
2004 <latexrelease> \prop_if_empty:cTF { g\_hook\_#1\_code\_prop }
2005 <latexrelease> { \_hook_log_line_indent:e { --- } }
2006 <latexrelease> {
2007 <latexrelease> \prop_map_inline:cn { g\_hook\_#1\_code\_prop }
2008 <latexrelease> { \_hook_log_line_indent:e
2009 <latexrelease> { ##1~>~\tl_to_str:n {##2} } }
2010 <latexrelease> }
2011 <latexrelease> \_hook_log_line:e
2012 <latexrelease> {
2013 <latexrelease> Document-level-(top-level)-code
2014 <latexrelease> \_hook_if_usable:nT {#1}
2015 <latexrelease> { ~(executed~
2016 <latexrelease> \_hook_if_reversed:nTF {#1} {first} {last} ) } :
2017 <latexrelease> }
2018 <latexrelease> \_hook_log_line_indent:e
2019 <latexrelease> {
2020 <latexrelease> \tl_if_empty:cTF { __hook_toplevel~#1 }
2021 <latexrelease> { --- }
2022 <latexrelease> { -> ~ \exp_args:Nv \tl_to_str:n
2023 <latexrelease> { __hook_toplevel~#1 } }
2024 <latexrelease> }
2025 <latexrelease> \_hook_log_line:e { Extra-code-for-next-invocation: }
2026 <latexrelease> \_hook_log_line_indent:e
2027 <latexrelease> {
2028 <latexrelease> \tl_if_empty:cTF { __hook_next~#1 }
2029 <latexrelease> { --- }
2030 <latexrelease> { ->~ \exp_args:Nv \_hook_log_next_code:n
2031 <latexrelease> { __hook_next~#1 } }
2032 <latexrelease> }
2033 <latexrelease> \_hook_log_line:e { Rules: }
2034 <latexrelease> \bool_set_true:N \l\_hook_tmpa_bool
2035 <latexrelease> \_hook_list_rules:nn {#1}
2036 <latexrelease> {
2037 <latexrelease> \bool_set_false:N \l\_hook_tmpa_bool
2038 <latexrelease> \_hook_log_line_indent:e
2039 <latexrelease> {
2040 <latexrelease> ##2~ with~
2041 <latexrelease> \str_if_eq:nnT {##3} {??} { default~ }
2042 <latexrelease> relation~ ##1
2043 <latexrelease> }
2044 <latexrelease> }
2045 <latexrelease> \bool_if:NT \l\_hook_tmpa_bool
2046 <latexrelease> { \_hook_log_line_indent:e { --- } }
2047 <latexrelease> \bool_lazy_and:nnTF
2048 <latexrelease> { \_hook_if_usable_p:n {#1} }
2049 <latexrelease> { ! \hook_if_empty_p:n {#1} }
2050 <latexrelease> {
2051 <latexrelease> \_hook_log_line:e
2052 <latexrelease> {
2053 <latexrelease> Execution-order
2054 <latexrelease> \bool_if:NTF \l\_hook_tmpa_bool

```

```

2055 <latexrelease> { \__hook_if_reversed:nT
2056 <latexrelease> {#1}{ ~(after-reversal) } }
2057 <latexrelease> { ~(after~
2058 <latexrelease> \__hook_if_reversed:nT {#1} { reversal~and~
2059 <latexrelease> applying~rules)
2060 <latexrelease> } :
2061 <latexrelease> }
2062 <latexrelease> #2 % \tl_show:n
2063 <latexrelease> {
2064 <latexrelease> \@spaces
2065 <latexrelease> \clist_if_empty:cTF { g__hook_#1_labels_clist }
2066 <latexrelease> { --- }
2067 <latexrelease> { \clist_use:cn
2068 <latexrelease> { g__hook_#1_labels_clist } { ,~ } }
2069 <latexrelease> }
2070 <latexrelease> }
2071 <latexrelease> {
2072 <latexrelease> \__hook_log_line:e { Execution-order: }
2073 <latexrelease> #2
2074 <latexrelease> {
2075 <latexrelease> \@spaces Not~set~because~the~hook~
2076 <latexrelease> \__hook_if_usable:nTF {#1}
2077 <latexrelease> { code~pool~is~empty }
2078 <latexrelease> { is~\__hook_if_disabled:nTF
2079 <latexrelease> {#1} {disabled} {undeclared} }
2080 <latexrelease> }
2081 <latexrelease> }
2082 <latexrelease> }
2083 <latexrelease> }
2084 <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 }₁2.

```

2085 <latexrelease> \IncludeInRelease{2023/06/01}{\__hook_log_next_code:n}
2086 <latexrelease> {Hooks~with~args}
__hook_log_next_code:n
2087 \exp_last_unbraced:NNNNo
2088 \cs_new:Npn \__hook_log_next_code:w #1 \c_right_brace_str { }
2089 <latexrelease> \EndIncludeInRelease
2090 <latexrelease> \IncludeInRelease{2020/10/01}{\__hook_log_next_code:n}
2091 <latexrelease> {Hooks~with~args}
2092 <latexrelease> \cs_gset:Npn \__hook_log_next_code:n #1
2093 <latexrelease> { \exp_args:No \tl_to_str:n { \use_none:nn #1 } }
2094 <latexrelease> \EndIncludeInRelease

```

Pretty-prints the number of arguments of a hook.

```

2095 \cs_new:Npn \__hook_print_args:nn #1 #2
2096 {
2097   \int_compare:nNnT {#2} > { 0 }
2098   {
2099     \__hook_if_declared:nT {#1} { \use_none:nnn }
2100     \__hook_if_cmd_hook:nT {#1}
2101     { \use_i:nnn { ~ (unknown ~ ) } }
2102     \use:n { ~ (#2 ~ ) }

```

```

2103         argument \int_compare:nNnT {#2} > { 1 } { s } )
2104     }
2105 }

```

(End of definition for `\hook_show:n` and others. These functions are documented on page 18.)

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

```

2106 \cs_new_protected:Npn \__hook_list_rules:nn #1 #2
2107 {
2108     \prop_if_exist:cT { g__hook_#1_code_prop }
2109     {
2110         \cs_set_protected:Npn \__hook_tmp:w ##1 ##2 ##3 {#2}
2111         \prop_map_inline:cn { g__hook_#1_code_prop }
2112         {
2113             \prop_map_inline:cn { g__hook_#1_code_prop }
2114             {
2115                 \__hook_if_label_case:nnnnn {##1} {####1}
2116                 { \prop_map_break: }
2117                 { \__hook_list_one_rule:nnn {##1} {####1} }
2118                 { \__hook_list_one_rule:nnn {####1} {##1} }
2119                 {#1}
2120             }
2121         }
2122     }
2123 }

```

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*.

```

2124 \cs_new_protected:Npn \__hook_list_one_rule:nnn #1#2#3
2125 {
2126     \__hook_list_if_rule_exists:nnnF {#1} {#2} {#3}
2127     { \__hook_list_if_rule_exists:nnnF {#1} {#2} { ?? } { } }
2128 }
2129 \cs_new_protected:Npn \__hook_list_if_rule_exists:nnnF #1#2#3
2130 {
2131     \if_cs_exist:w g__hook_#3_rule_#1 | #2 _t1 \cs_end:
2132     \exp_args:Nv \__hook_tmp:w
2133     { g__hook_#3_rule_#1 | #2 _t1 } { #1 | #2 } {#3}
2134     \exp_after:wN \use_none:nn
2135     \fi:
2136     \use:n
2137 }

```

(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`.

```

2138 \cs_new_protected:Npn \__hook_debug_print_rules:n #1
2139 {
2140     \iow_term:n { The~hook~#1~contains~the~rules: }
2141     \cs_set_protected:Npn \__hook_tmp:w ##1

```

```

2142     {
2143       \__hook_list_rules:nn {#1}
2144       {
2145         \iow_term:e
2146         {
2147           > ##1 {####2} ##1 => ##1 {####1}
2148           \str_if_eq:nnT {####3} {??} { ~(default) }
2149         }
2150       }
2151     }
2152     \exp_args:No \__hook_tmp:w { \use:nn { ~ } { ~ } }
2153   }

```

(End of definition for __hook_debug_print_rules:n.)

4.8 Specifying code for next invocation

`\hook_gput_next_code:nn`

```

2154 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_gput_next_code:nn}
2155 <latexrelease> {Hooks~with~args}
2156 \cs_new_protected:Npn \hook_gput_next_code:nn #1 #2
2157 {
2158   \__hook_replacing_args_false:
2159   \__hook_normalize_hook_args:Nn \__hook_gput_next_code:nn {#1} {#2}
2160
2161   \__hook_debug:n{\iow_term:e{[lthooks]~ Add~ to~
2162     \__hook_if_usable:nF {#1} { undeclared~ }
2163     hook~ ' #1 '~ ( next~ invocation~ only )
2164     \on@line
2165     ^^J [lthooks] \@spaces
2166     <-- \tl_to_str:n{#2}
2167   }
2168   \__hook_replacing_args_reset:
2169 }
2170 \cs_new_protected:Npn \hook_gput_next_code_with_args:nn #1 #2
2171 {
2172   \__hook_replacing_args_true:
2173   \__hook_normalize_hook_args:Nn \__hook_gput_next_code:nn {#1} {#2}
2174
2175   \__hook_debug:n{\iow_term:e{[lthooks]~ Add~ to~
2176     \__hook_if_usable:nF {#1} { undeclared~ }
2177     hook~ ' #1 '~ ( next~ invocation~ only )
2178     \on@line
2179     ^^J [lthooks] \@spaces
2180     <-- \tl_to_str:n{#2}
2181   }
2182   \__hook_replacing_args_reset:
2183 }
2184 <latexrelease>\EndIncludeInRelease
2185 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_gput_next_code:nn}
2186 <latexrelease> {Hooks~with~args}
2187 <latexrelease>\cs_gset_protected:Npn \hook_gput_next_code:nn #1

```

```

2188 <latexrelease> { \__hook_normalize_hook_args:Nn
2189 <latexrelease> \__hook_gput_next_code:nn {#1} }
2190 <latexrelease>\cs_gset_protected:Npn \hook_gput_next_code_with_args:nn #1 #2 { }
2191 <latexrelease>\EndIncludeInRelease

```

(End of definition for \hook_gput_next_code:nn. This function is documented on page 17.)

__hook_gput_next_code:nn

```

2192 \cs_new_protected:Npn \__hook_gput_next_code:nn #1 #2
2193 {
2194   \__hook_if_disabled:nTF {#1}
2195     { \msg_error:nnn { hooks } { hook-disabled } {#1} }
2196     {
2197       \__hook_if_structure_exist:nTF {#1}
2198         { \__hook_gput_next_do:nn }
2199         { \__hook_try_declaring_generic_next_hook:nn }
2200           {#1} {#2}
2201     }
2202 }

```

(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.

```

2203 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_gput_next_do:nn}
2204 <latexrelease> {Hooks~with~args}
2205 \cs_new_protected:Npn \__hook_gput_next_do:nn #1
2206 {
2207   \__hook_init_structure:n {#1}
2208   \__hook_chk_args_allowed:nn {#1} { AddToHookNext }
2209   \__hook_cs_if_empty:cT { __hook~#1 }
2210     { \__hook_update_hook_code:n {#1} }
2211   \__hook_cs_if_empty:cT { __hook_next~#1 }
2212     { \__hook_next_gset:nn {#1} { \__hook_clear_next:n {#1} } }
2213   \__hook_cs_gput_right:nnn { _next } {#1}
2214 }
2215 <latexrelease>\EndIncludeInRelease
2216 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_gput_next_do:nn}
2217 <latexrelease> {Hooks~with~args}
2218 <latexrelease>\cs_gset_protected:Npn \__hook_gput_next_do:nn #1
2219 <latexrelease> {
2220 <latexrelease> \exp_args:Nc \__hook_gput_next_do:Nnn
2221 <latexrelease> { __hook_next~#1 } {#1}
2222 <latexrelease> }
2223 <latexrelease>\cs_gset_protected:Npn \__hook_gput_next_do:Nnn #1 #2
2224 <latexrelease> {
2225 <latexrelease> \tl_if_empty:cT { __hook~#2 }
2226 <latexrelease> { \__hook_update_hook_code:n {#2} }
2227 <latexrelease> \tl_if_empty:NT #1

```

```

2228 <latexrelease>      { \_hook_tl_gset:Nn #1 { \_hook_clear_next:n {#2} } }
2229 <latexrelease>      \_hook_tl_gput_right:Nn #1
2230 <latexrelease>      }
2231 <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.

```

2232 \cs_new_protected:Npn \hook_gclear_next_code:n #1
2233 { \_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 17.)

_hook_clear_next:n

```

2234 <latexrelease>\IncludeInRelease{2023/06/01}{\_hook_clear_next:n}
2235 <latexrelease>      {Hooks~with~args}
2236 \cs_new_protected:Npn \_hook_clear_next:n #1
2237 { \_hook_next_gset:nn {#1} { } }
2238 <latexrelease>\EndIncludeInRelease
2239 <latexrelease>\IncludeInRelease{2020/10/01}{\_hook_clear_next:n}
2240 <latexrelease>      {Hooks~with~args}
2241 <latexrelease>\cs_gset_protected:Npn \_hook_clear_next:n #1
2242 <latexrelease> { \cs_gset_eq:cN { \_hook_next~#1 } \c_empty_tl }
2243 <latexrelease>\EndIncludeInRelease

```

(End of definition for _hook_clear_next:n.)

4.9 Using the hook

\hook_use: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 T_EX low-level \csname ... \endcsname construct \tl_if_exist:c is careful to avoid this.

```

2244 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_use:n}
2245 <latexrelease>      {Hooks~with~args}
2246 \cs_new_protected:Npn \hook_use:n #1
2247 {
2248   \_hook_preamble_hook:n {#1}
2249   \_hook_use_initialized:n {#1}
2250 }
2251 \cs_new:Npn \_hook_use_initialized:n #1

```

```

2252 {
2253   \if_cs_exist:w __hook~#1 \cs_end:
2254   \cs:w __hook~#1 \use_i:nn
2255   \fi:
2256   \use_none:n
2257   \cs_end:
2258 }
2259 \cs_new_protected:Npn \__hook_preamble_hook:n #1
2260 {
2261   \if_cs_exist:w __hook~#1 \cs_end:
2262   \__hook_initialize_hook_code:n {#1}
2263   \fi:
2264 }
2265 \<latexrelease>\EndIncludeInRelease

2266 \<latexrelease>\IncludeInRelease{2021/11/15}{\hook_use:n}
2267 \<latexrelease>{Standardize~generic~hook~names}
2268 \<latexrelease>\cs_new_protected:Npn \hook_use:n #1
2269 \<latexrelease> {
2270   \<latexrelease> \tl_if_exist:cT { __hook~#1 }
2271   \<latexrelease> {
2272     \<latexrelease> \__hook_preamble_hook:n {#1}
2273     \<latexrelease> \cs:w __hook~#1 \cs_end:
2274   \<latexrelease> }
2275 \<latexrelease> }
2276 \<latexrelease>\cs_new:Npn \__hook_use_initialized:n #1
2277 \<latexrelease> {
2278   \<latexrelease> \if_cs_exist:w __hook~#1 \cs_end:
2279   \<latexrelease> \cs:w __hook~#1 \exp_after:wN \cs_end:
2280   \<latexrelease> \fi:
2281 \<latexrelease> }
2282 \<latexrelease>\cs_new_protected:Npn \__hook_preamble_hook:n #1
2283 \<latexrelease> { \__hook_initialize_hook_code:n {#1} }
2284 \<latexrelease>\cs_new:Npn \hook_use:nnw #1 { }
2285 \<latexrelease>\EndIncludeInRelease

2286 \<latexrelease>\IncludeInRelease{2020/10/01}{\hook_use:n}
2287 \<latexrelease>{Standardize~generic~hook~names}
2288 \<latexrelease>\cs_new_protected:Npn \hook_use:n #1
2289 \<latexrelease> {
2290   \<latexrelease> \tl_if_exist:cTF { __hook~#1 }
2291   \<latexrelease> {
2292     \<latexrelease> \__hook_preamble_hook:n {#1}
2293     \<latexrelease> \cs:w __hook~#1 \cs_end:
2294   \<latexrelease> }
2295   \<latexrelease> { \__hook_use:wn #1 / \s__hook_mark {#1} }
2296 \<latexrelease> }
2297 \<latexrelease>\cs_new:Npn \__hook_use_initialized:n #1
2298 \<latexrelease> {
2299   \<latexrelease> \if_cs_exist:w __hook~#1 \cs_end:
2300   \<latexrelease> \else:
2301   \<latexrelease> \__hook_use_undefined:w
2302   \<latexrelease> \fi:
2303   \<latexrelease> \cs:w __hook~#1 \__hook_use_end:
2304 \<latexrelease> }

```

```

2305 <latexrelease>\cs_new:Npn \__hook_use_undefined:w
2306 <latexrelease> #1 #2 __hook~#3 \__hook_use_end:
2307 <latexrelease> {
2308 <latexrelease> #1 % fi
2309 <latexrelease> \__hook_use:wn #3 / \s__hook_mark {#3}
2310 <latexrelease> }
2311 <latexrelease>\cs_new_protected:Npn \__hook_preamble_hook:n #1
2312 <latexrelease> { \__hook_initialize_hook_code:n {#1} }
2313 <latexrelease>\cs_new_eq:NN \__hook_use_end: \cs_end:
2314 <latexrelease>\cs_new:Npn \hook_use:nnw #1 { }
2315 <latexrelease>\EndIncludeInRelease

```

(End of definition for \hook_use:n, __hook_use_initialized:n, and __hook_preamble_hook:n. This function is documented on page 16.)

\hook_use:nnw

__hook_use_initialized:nnw

```

2316 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_use:nnw}
2317 <latexrelease> {Hooks~with~args}
2318 \cs_new_protected:Npn \hook_use:nnw #1
2319 {
2320 \__hook_preamble_hook:n {#1}
2321 \__hook_use_initialized:nnw {#1}
2322 }
2323 \cs_new:Npn \__hook_use_initialized:nnw #1 #2
2324 {
2325 \cs:w
2326 \if_cs_exist:w __hook~#1 \cs_end:
2327 __hook~#1
2328 \else:
2329 use_none: \prg_replicate:nn {#2} { n }
2330 \fi:
2331 \cs_end:
2332 }
2333 <latexrelease>\EndIncludeInRelease
2334 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_use:nnw}
2335 <latexrelease> {Hooks~with~args}
2336 <latexrelease>\cs_gset:Npn \hook_use:nnw #1 #2
2337 <latexrelease> { \use:c { use_none: \prg_replicate:nn {#2} { n } } }
2338 <latexrelease>\EndIncludeInRelease

```

(End of definition for \hook_use:nnw and __hook_use_initialized:nnw. This function is documented on page 16.)

__hook_post_initialization_defs:

```

2339 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_post_initialization_defs:}
2340 <latexrelease> {Hooks~with~args}
2341 \cs_new_protected:Npn \__hook_post_initialization_defs:
2342 {
2343 \cs_gset_eq:NN \hook_use:n \__hook_use_initialized:n
2344 \cs_gset_eq:NN \hook_use:nnw \__hook_use_initialized:nnw
2345 \cs_gset_eq:NN \__hook_preamble_hook:n \use_none:n
2346 \cs_gset_eq:NN \__hook_post_initialization_defs: \prg_do_nothing:
2347 }
2348 <latexrelease>\EndIncludeInRelease
2349 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_post_initialization_defs:}

```

```

2350 <latexrelease> {Hooks~with~args}
2351 <latexrelease>\cs_undefine:N \__hook_post_initialization_defs:
2352 <latexrelease>\EndIncludeInRelease

```

(End of definition for __hook_post_initialization_defs:.)

__hook_use:wn __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.

```

2353 <latexrelease>\IncludeInRelease{2021/11/15}{\__hook_use:wn}
2354 <latexrelease> {Standardize~generic~hook~names}
2355 <latexrelease>\EndIncludeInRelease
2356 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_use:wn}
2357 <latexrelease> {Standardize~generic~hook~names}
2358 <latexrelease>\cs_new:Npn \__hook_use:wn #1 / #2 \s_hook_mark #3
2359 <latexrelease> {
2360 <latexrelease> \str_if_eq:nnTF {#1} { file }
2361 <latexrelease> { \__hook_try_file_hook:n {#3} }
2362 <latexrelease> { } % Hook doesn't exist
2363 <latexrelease> }

2364 <latexrelease>\cs_new_protected:Npn \__hook_try_file_hook:n #1
2365 <latexrelease> {
2366 <latexrelease> \__hook_if_file_hook:wTF #1 / / \s_hook_mark
2367 <latexrelease> {
2368 <latexrelease> \exp_args:Ne \__hook_if_usable_use:n
2369 <latexrelease> { \exp_args:Ne \__hook_file_hook_normalize:n {#1} }
2370 <latexrelease> }
2371 <latexrelease> { \__hook_if_usable_use:n {#1} }
2372 <latexrelease> % file/ generic hook (e.g. file/before)
2373 <latexrelease> }

2374 <latexrelease>\cs_new_protected:Npn \__hook_if_usable_use:n #1
2375 <latexrelease> {
2376 <latexrelease> \tl_if_exist:cT { __hook~#1 }
2377 <latexrelease> {
2378 <latexrelease> \__hook_preamble_hook:n {#1}
2379 <latexrelease> \cs:w __hook~#1 \cs_end:
2380 <latexrelease> }
2381 <latexrelease> }
2382 <latexrelease>\EndIncludeInRelease

```

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

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

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

```

2383 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_use_once:nnw}
2384 <latexrelease> {Hooks~with~args}
2385 \cs_new_protected:Npn \hook_use_once:n #1
2386 {
2387   \__hook_if_execute_immediately:nF {#1}
2388   { \__hook_normalize_hook_args:Nn \__hook_use_once:nn
2389     { \use:n {#1} } { 0 } }
2390 }
2391 \cs_new_protected:Npn \hook_use_once:nnw #1 #2
2392 {
2393   \__hook_if_execute_immediately:nF {#1}
2394   { \__hook_normalize_hook_args:Nn \__hook_use_once:nn
2395     { \use:n {#1} } {#2} }
2396 }
2397 <latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_use_once:n` and `\hook_use_once:nnw`. These functions are documented on page 16.)

```

2398 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_use_once:nnw}
2399 <latexrelease> {Hooks~with~args}
2400 <latexrelease>\cs_gset_protected:Npn \hook_use_once:n #1
2401 <latexrelease> {
2402 <latexrelease>   \__hook_if_execute_immediately:nF {#1}
2403 <latexrelease>   { \__hook_normalize_hook_args:Nn \__hook_use_once:n
2404 <latexrelease>     { \use:n {#1} } }
2405 <latexrelease> }
2406 <latexrelease>\cs_gset:Npn \hook_use_once:nnw #1 #2
2407 <latexrelease> { \use:c { use_none: \prg_replicate:nn {#2} { n } } }
2408 <latexrelease>\EndIncludeInRelease

```

`\__hook_use_once:nn`

```

2409 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_use_once:nn}
2410 <latexrelease> {Hooks~with~args}
2411 \cs_new_protected:Npn \__hook_use_once:nn #1 #2
2412 {
2413   \__hook_preamble_hook:n {#1}
2414   \__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.

```

2415   \__hook_replacing_args_false:
2416   \__hook_cs_gput_right:nnn { _next } {#1}
2417   { \__hook_use_once_clear:n {#1} }
2418   \__hook_replacing_args_reset:
2419   \__hook_if_usable:nTF {#1}
2420   { \__hook_use_initialized:n {#1} }
2421   {
2422     \int_compare:nNnT {#2} > { 0 }
2423     { \use:c { use_none: \prg_replicate:nn {#2} { n } } }
2424   }

```

```

2425 }
2426 \<latexrelease>\EndIncludeInRelease
2427 %
2428 \<latexrelease>\IncludeInRelease{2020/10/01}{\__hook_use_once:nn}
2429 \<latexrelease>{Hooks~with~args}
2430 \<latexrelease>\cs_gset_protected:Npn \__hook_use_once:n #1
2431 \<latexrelease> {
2432 \<latexrelease> \__hook_preamble_hook:n {#1}
2433 \<latexrelease> \__hook_use_once_set:n {#1}
2434 \<latexrelease> \__hook_use_initialized:n {#1}
2435 \<latexrelease> \__hook_use_once_clear:n {#1}
2436 \<latexrelease> }
2437 \<latexrelease>\cs_undefine:N \__hook_use_once:nn
2438 \<latexrelease>\EndIncludeInRelease

```

(End of definition for __hook_use_once:nn.)

__hook_use_once_set: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.

```

2439 \<latexrelease>\IncludeInRelease{2023/06/01}{\__hook_use_once_clear:n}
2440 \<latexrelease>{Hooks~with~args}
2441 \cs_new_protected:Npn \__hook_use_once_set:n #1
2442 { \__hook_tl_gset:cn { g__hook_#1_reversed_tl } { I } }
2443 \cs_new_protected:Npn \__hook_use_once_clear:n #1
2444 {
2445 \__hook_code_gset:nn {#1} { }
2446 \__hook_next_gset:nn {#1} { }
2447 \__hook_toplevel_gset:nn {#1} { }
2448 \prop_gclear_new:c { g__hook_#1_code_prop }
2449 }
2450 \<latexrelease>\EndIncludeInRelease
2451 \<latexrelease>\IncludeInRelease{2020/10/01}{\__hook_use_once_clear:n}
2452 \<latexrelease>{Hooks~with~args}
2453 \<latexrelease>\cs_new_protected:Npn \__hook_use_once_clear:n #1
2454 \<latexrelease> {
2455 \<latexrelease> \__hook_tl_gclear:c { __hook~#1 }
2456 \<latexrelease> \__hook_tl_gclear:c { __hook_next~#1 }
2457 \<latexrelease> \__hook_tl_gclear:c { __hook_toplevel~#1 }
2458 \<latexrelease> \prop_gclear_new:c { g__hook_#1_code_prop }
2459 \<latexrelease> }
2460 \<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.

```

2461 \prg_new_conditional:Npnn \__hook_if_execute_immediately:n #1 { T, F, TF }
2462 {
2463 \exp_after:wN \__hook_use_none_delimit_by_s_mark:w
2464 \if:w I

```

```

2465         \if_cs_exist:w g__hook_#1_reversed_tl \cs_end:
2466         \cs:w g__hook_#1_reversed_tl \exp_after:wN \cs_end:
2467         \fi:
2468         X
2469         \s__hook_mark \prg_return_true:
2470     \else:
2471         \s__hook_mark \prg_return_false:
2472     \fi:
2473 }

```

(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 $\langle hook \rangle$ being empty means that all three of its `\g__hook_⟨hook⟩_code_prop`, its `\__hook_toplevel_⟨hook⟩` and its `\__hook_next_⟨hook⟩` are empty.

```

2474 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_if_empty:n}
2475 <latexrelease>           {Hooks-with-args}
2476 \prg_new_conditional:Npnn \hook_if_empty:n #1 { p , T , F , TF }
2477 {
2478     \if:w
2479         T
2480         \prop_if_exist:cT { g__hook_#1_code_prop }
2481         { \prop_if_empty:cF { g__hook_#1_code_prop } { F } }
2482         \__hook_cs_if_empty:cF { __hook_toplevel~#1 } { F }
2483         \__hook_cs_if_empty:cF { __hook_next~#1 } { F }
2484         T
2485         \prg_return_true:
2486     \else:
2487         \prg_return_false:
2488     \fi:
2489 }
2490 <latexrelease>\EndIncludeInRelease

2491 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_if_empty:n}
2492 <latexrelease>           {Hooks-with-args}
2493 <latexrelease>\prg_new_conditional:Npnn \hook_if_empty:n #1 { p , T , F , TF }
2494 <latexrelease> {

```

```

2495 <latexrelease> \__hook_if_structure_exist:nTF {#1}
2496 <latexrelease> {
2497 <latexrelease> \bool_lazy_and:nnTF
2498 <latexrelease> { \prop_if_empty_p:c { g__hook_#1_code_prop } }
2499 <latexrelease> {
2500 <latexrelease> \bool_lazy_and_p:nn
2501 <latexrelease> { \tl_if_empty_p:c { __hook_toplevel~#1 } }
2502 <latexrelease> { \tl_if_empty_p:c { __hook_next~#1 } }
2503 <latexrelease> }
2504 <latexrelease> { \prg_return_true: }
2505 <latexrelease> { \prg_return_false: }
2506 <latexrelease> }
2507 <latexrelease> { \prg_return_true: }
2508 <latexrelease> }
2509 <latexrelease> \EndIncludeInRelease

```

(End of definition for \hook_if_empty:nTF. This function is documented on page 18.)

__hook_if_usable_p:n 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).

```

2510 \prg_new_conditional:Npnn \__hook_if_usable:n #1 { p , T , F , TF }
2511 {
2512 \cs_if_exist:cTF { __hook~#1 }
2513 { \prg_return_true: }
2514 { \prg_return_false: }
2515 }

```

(End of definition for __hook_if_usable:nTF.)

__hook_if_structure_exist_p:n 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.

```

2516 \prg_new_conditional:Npnn \__hook_if_structure_exist:n #1 { p , T , F , TF }
2517 {
2518 \prop_if_exist:cTF { g__hook_#1_code_prop }
2519 { \prg_return_true: }
2520 { \prg_return_false: }
2521 }

```

(End of definition for __hook_if_structure_exist:nTF.)

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

```

\__hook_if_declared:nTF
2522 \prg_new_conditional:Npnn \__hook_if_declared:n #1 { p , T , F , TF }
2523 {
2524 \tl_if_exist:cTF { g__hook_#1_declared_tl }
2525 { \prg_return_true: }
2526 { \prg_return_false: }
2527 }

```

(End of definition for __hook_if_declared:nTF.)

`\_hook_if_reversed_p:n` An internal conditional that checks if a hook is reversed.

`\_hook_if_reversed:nTF`

```

2528 \prg_new_conditional:Npnn \_hook_if_reversed:n #1 { p , T , F , TF }
2529 {
2530   \exp_after:wN \_hook_use_none_delimit_by_s_mark:w
2531   \if:w - \cs:w g__hook_#1_reversed_tl \cs_end:
2532     \s__hook_mark \prg_return_true:
2533   \else:
2534     \s__hook_mark \prg_return_false:
2535   \fi:
2536 }

```

(End of definition for `\_hook_if_reversed:nTF`.)

`\_hook_if_generic_p:n` 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.

`\_hook_if_generic:nTF`

`\_hook_if_deprecated_generic_p:n`

`\_hook_if_deprecated_generic:nTF`

```

2537 \prg_new_conditional:Npnn \_hook_if_generic:n #1 { T , TF }
2538 { \_hook_if_generic:w #1 / / / \s__hook_mark }
2539 \cs_new:Npn \_hook_if_generic:w #1 / #2 / #3 / #4 \s__hook_mark
2540 {
2541   \cs_if_exist:cTF { c__hook_generic_#1/./#3_tl }
2542     { \prg_return_true: }
2543     { \prg_return_false: }
2544 }
2545 \prg_new_conditional:Npnn \_hook_if_deprecated_generic:n #1 { T , TF }
2546 { \_hook_if_deprecated_generic:w #1 / / / \s__hook_mark }
2547 \cs_new:Npn \_hook_if_deprecated_generic:w #1 / #2 / #3 / #4 \s__hook_mark
2548 {
2549   \cs_if_exist:cTF { c__hook_deprecated_#1/./#2_tl }
2550     {
2551       \tl_if_empty:nTF {#3}
2552         { \prg_return_false: }
2553         { \prg_return_true: }
2554     }
2555     { \prg_return_false: }
2556 }

```

(End of definition for `\_hook_if_generic:nTF` and `\_hook_if_deprecated_generic:nTF`.)

`\_hook_if_cmd_hook_p:n` An internal conditional that checks if a given hook is a valid generic cmd hook.

`\_hook_if_cmd_hook:nTF`

`\_hook_if_cmd_hook_p:w`

`\_hook_if_cmd_hook:wTF`

```

2557 <latexrelease>\IncludeInRelease{2023/06/01}{\_hook_if_cmd_hook:n}
2558 <latexrelease>{Hooks~with~args}
2559 \prg_new_conditional:Npnn \_hook_if_cmd_hook:n #1 { T }
2560 { \_hook_if_cmd_hook:w #1 / / / \s__hook_mark }
2561 \cs_new:Npn \_hook_if_cmd_hook:w #1 / #2 / #3 / #4 \s__hook_mark
2562 {
2563   \if:w Y
2564     \str_if_eq:nnF {#1} { cmd } { N }
2565     \tl_if_exist:cF { c__hook_generic_#1/./#3_tl } { N }
2566     Y
2567     \prg_return_true:
2568   \else:
2569     \prg_return_false:
2570   \fi:
2571 }

```

```

2572 <latexrelease>\EndIncludeInRelease
2573 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_if_cmd_hook:n}
2574 <latexrelease>{Hooks~with~args}
2575 <latexrelease>\cs_undefine:N \__hook_if_cmd_hook:nT
2576 <latexrelease>\EndIncludeInRelease

```

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

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

```

\__hook_if_generic_reversed:nTF
2577 \prg_new_conditional:Npnn \__hook_if_generic_reversed:n #1 { T }
2578 { \__hook_if_generic_reversed:w #1 / / \scan_stop: }
2579 \cs_new:Npn \__hook_if_generic_reversed:w #1 / #2 / #3 / #4 \scan_stop:
2580 {
2581   \if_charcode:w - \cs:w c__hook_generic_#1/./#3_t1 \cs_end:
2582   \prg_return_true:
2583   \else:
2584   \prg_return_false:
2585   \fi:
2586 }

```

(End of definition for __hook_if_generic_reversed:nTF.)

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

```

\__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
2587 \seq_new:N \g__hook_replacing_stack_seq
2588 \cs_new:Npn \__hook_misused_if_replacing_args:nn #1 #2
2589 {
2590   \msg_expandable_error:nnn { latex2e } { should-not-happen }
2591   { Misused~\__hook_if_replacing_args:. }
2592 }
2593 \cs_new:Npn \__hook_if_replacing_args:TF
2594 { \__hook_misused_if_replacing_args:nn }
2595 \cs_new_protected:Npn \__hook_replacing_args_true:
2596 {
2597   \seq_gpush:No \g__hook_replacing_stack_seq
2598   { \__hook_if_replacing_args:TF }
2599   \cs_set:Npn \__hook_if_replacing_args:TF { \use_i:nn }
2600 }
2601 \cs_new_protected:Npn \__hook_replacing_args_false:
2602 {
2603   \seq_gpush:No \g__hook_replacing_stack_seq
2604   { \__hook_if_replacing_args:TF }
2605   \cs_set:Npn \__hook_if_replacing_args:TF { \use_ii:nn }
2606 }
2607 \cs_new_protected:Npn \__hook_replacing_args_reset:
2608 {
2609   \seq_gpop:NN \g__hook_replacing_stack_seq \l__hook_return_tl
2610   \cs_gset_eq:NN \__hook_if_replacing_args:TF \l__hook_return_tl
2611 }

```

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

4.11 Messages

Hook errors are LaTeX kernel errors:

```
2612 \prop_gput:Nnn \g_msg_module_type_prop { hooks } { LaTeX }
```

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

```
2613 \prop_gput:Nnn \g_msg_module_type_prop { latex2e } { LaTeX }
```

```
2614 \prop_gput:Nnn \g_msg_module_name_prop { latex2e } { kernel }
```

```
2615 \msg_new:nnnn { hooks } { labels-incompatible }
```

```
2616 {
2617   Labels~'#1'~and~'#2'~are~incompatible
2618   \str_if_eq:nnF {#3} {??} { ~in-hook~'#3' } .~
2619   \int_compare:nNnTF {#4} = { 1 }
2620     { The~ code~ for~ both~ labels~ will~ be~ dropped. }
2621     { You~ may~ see~ errors~ later. }
2622 }
2623 { LaTeX~found~two~incompatible~labels~in~the~same~hook.~
2624   This~indicates~an~incompatibility~between~packages. }
```

```
2625 \msg_new:nnnn { hooks } { exists }
```

```
2626 { Hook~'#1'~ has~ already~ been~ declared. }
2627 { There~ already~ exists~ a~ hook~ declaration~ with~ this~
2628   name.\\
2629   Please~ use~ a~ different~ name~ for~ your~ hook. }
```

```
2630 <latexrelease>\IncludeInRelease{2023/06/01}{too-many-args}
```

```
2631 <latexrelease>{Hooks-with-args}
```

```
2632 \msg_new:nnnn { hooks } { too-many-args }
```

```
2633 { Too-many~arguments~for~hook~'#1'. }
2634 {
2635   You~tried~to~declare~a~hook~with~#2~arguments,~but~a~
2636   hook~can~only~have~up~to~nine.~LaTeX~will~define~this~
2637   hook~with~nine~arguments.
2638 }
```

```
2639 \msg_new:nnnn { hooks } { without-args }
```

```
2640 { Hook~'#1'~has~no~arguments. }
2641 {
2642   You~tried~to~use~\iow_char:N\\#2WithArguments~
2643   on~a~hook~that~takes~no~arguments.\\
2644   Check~the~usage~of~the~hook~or~use~\iow_char:N\\#2~instead.\\
2645   \\
2646   LaTeX~will~use~\iow_char:N\\#2.
2647 }
```

```
2648 \msg_new:nnnn { hooks } { one-time-args }
```

```
2649 { You~can't~have~arguments~in~used~one-time~hook~'#1'. }
2650 {
2651   You~tried~to~use~\iow_char:N\\#2WithArguments~
2652   on~a~one-time~hook~that~has~already~been~used.~
2653   You~have~to~add~the~code~before~the~hook~is~used,~
2654   or~add~the~code~without~arguments~using~\iow_char:N\\#2~instead.\\
2655   \\
2656   LaTeX~will~use~\iow_char:N\\#2.
2657 }
```

```

2658 <latexrelease>\EndIncludeInRelease
2659 <latexrelease>\IncludeInRelease{2020/10/01}{too-many-args}
2660 <latexrelease>{Hooks-with-args}
2661 <latexrelease>\EndIncludeInRelease

2662 \msg_new:nnnn { hooks } { hook-disabled }
2663 { Cannot~add~code~to~disabled~hook~'#1'. }
2664 {
2665   The~hook~'#1'~you~tried~to~add~code~to~was~previously~disabled~
2666   with~\iow_char:N\hook_disable_generic:n~or~
2667   \iow_char:N\DisableGenericHook,~so~
2668   it~cannot~have~code~added~to~it.
2669 }

2670 \msg_new:nnn { hooks } { empty-label }
2671 {
2672   Empty~code~label~\msg_line_context:~
2673   Using~'\_hook_currname_or_default:'~instead.
2674 }

2675 \msg_new:nnn { hooks } { empty-hook }
2676 {
2677   Empty~hook~name~\msg_line_context:.
2678 }

2679 \msg_new:nnn { hooks } { no-default-label }
2680 {
2681   Missing~(empty)~default~label~\msg_line_context:. \\
2682   This~command~was~ignored.
2683 }

2684 \msg_new:nnnn { hooks } { unknown-rule }
2685 {
2686   Unknown~ relationship~ '#3'~
2687   between~ labels~ '#2'~ and~ '#4'~
2688   \str_if_eq:nnF {#1} {??} { ~in~hook~'#1' }. ~
2689   Perhaps~ a~ misspelling?
2690 }
2691 {
2692   The~ relation~ used~ not~ known~ to~ the~ system.~ Allowed~ values~ are~
2693   'before'~ or~ '<',~
2694   'after'~ or~ '>',~
2695   'incompatible-warning',~
2696   'incompatible-error',~
2697   'voids'~ or~
2698   'unrelated'.
2699 }

2700 \msg_new:nnnn { hooks } { rule-too-late }
2701 {
2702   Sorting~rule~for~'#1'~hook~applied~too~late.\\
2703   Try~setting~this~rule~earlier.
2704 }
2705 {
2706   You~tried~to~set~the~ordering~of~hook~'#1'~using\\
2707   \ \ \iow_char:N\DeclareHookRule{#1}{#2}{#3}{#4}\\
2708   but~hook~'#1'~was~already~used~as~a~one~time~hook,~

```

```

2709     thus~sorting~is~\
2710     no~longer~possible.~Declare~the~rule~
2711     before~the~hook~is~used.
2712 }
2713 \msg_new:nnnn { hooks } { misused-top-level }
2714 {
2715     Illegal~use~of~\iow_char:N \AddToHook{#1}[top-level]{...}.\
2716     'top-level'~is~reserved~for~the~user's~document.
2717 }
2718 {
2719     The~'top-level'~label~is~meant~for~user~code~only,~and~should~only~
2720     be~used~(sparingly)~in~the~main~document.~Use~the~default~label~
2721     '\_hook_currname_or_default:'~for~this~\@cls@pkg,~or~another~
2722     suitable~label.
2723 }
2724 \msg_new:nnn { hooks } { set-top-level }
2725 {
2726     You~cannot~change~the~default~label~#1~'top-level'.~Illegal \
2727     \use:nn { ~ } { ~ } \iow_char:N \#2{#3} \
2728     \msg_line_context:.
2729 }
2730 \msg_new:nnn { hooks } { extra-pop-label }
2731 {
2732     Extra~\iow_char:N \PopDefaultHookLabel. \
2733     This~command~will~be~ignored.
2734 }
2735 \msg_new:nnn { hooks } { missing-pop-label }
2736 {
2737     Missing~\iow_char:N \PopDefaultHookLabel. \
2738     The~label~'#1'~was~pushed~but~never~popped.~Something~is~wrong.
2739 }
2740 \msg_new:nnn { latex2e } { should-not-happen }
2741 {
2742     This~should~not~happen.~#1 \
2743     Please~report~at~https://github.com/latex3/latex2e.
2744 }
2745 \msg_new:nnn { hooks } { activate-disabled }
2746 {
2747     Cannot~ activate~ hook~ '#1'~ because~ it~ is~ disabled!
2748 }
2749 \msg_new:nnn { hooks } { cannot-remove }
2750 {
2751     Cannot~remove~chunk~'#2'~from~hook~'#1'~because~
2752     \_hook_if_structure_exist:nTF {#1}
2753     { it~does~not~exist~in~that~hook. }
2754     { the~hook~does~not~exist. }
2755 }
2756 \msg_new:nnn { hooks } { generic-deprecated }
2757 {
2758     Generic~hook~'#1/#2/#3'~is~deprecated. \
2759     Use~hook~'#1/#3/#2'~instead.
2760 }

```

4.12 L^AT_EX 2_ε package interface commands

\NewHook Declaring new hooks ...

```

\NewReversedHook 2761 \NewDocumentCommand \NewHook          { m }
\NewMirroredHookPair 2762 { \hook_new:n {#1} }
2763 \NewDocumentCommand \NewReversedHook    { m }
2764 { \hook_new_reversed:n {#1} }
2765 \NewDocumentCommand \NewMirroredHookPair { mm }
2766 { \hook_new_pair:nn {#1}{#2} }
```

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

\NewHookWithArguments Declaring new hooks with arguments...

```

\NewReversedHookWithArguments 2767 <latexrelease>\IncludeInRelease{2023/06/01}{\NewHookWithArguments}
\NewMirroredHookPairWithArguments 2768 <latexrelease>          {Hooks~with~args}
2769 \NewDocumentCommand \NewHookWithArguments          { mm }
2770 { \hook_new_with_args:nn {#1} {#2} }
2771 \NewDocumentCommand \NewReversedHookWithArguments  { mm }
2772 { \hook_new_reversed_with_args:nn {#1} {#2} }
2773 \NewDocumentCommand \NewMirroredHookPairWithArguments { mmm }
2774 { \hook_new_pair_with_args:nnn {#1} {#2} {#3} }
2775 <latexrelease>\EndIncludeInRelease
2776 <latexrelease>\IncludeInRelease{2020/10/01}{\NewHookWithArguments}
2777 <latexrelease>          {Hooks~with~args}
2778 <latexrelease>\cs_new_protected:Npn \NewHookWithArguments #1 #2 { }
2779 <latexrelease>\cs_new_protected:Npn \NewReversedHookWithArguments #1 #2 { }
2780 <latexrelease>\cs_new_protected:Npn \NewMirroredHookPairWithArguments #1 #2 #3{ }
2781 <latexrelease>\EndIncludeInRelease
```

(End of definition for \NewHookWithArguments, \NewReversedHookWithArguments, and \NewMirroredHookPairWithArgument. These functions are documented on page 3.)

```

2782 <latexrelease>\IncludeInRelease{2021/06/01}{\ActivateGenericHook}
2783 <latexrelease>          {Providing~hooks}
```

\ActivateGenericHook Providing new hooks ...

```

2784 \NewDocumentCommand \ActivateGenericHook { m }
2785 { \hook_activate_generic:n {#1} }
```

(End of definition for \ActivateGenericHook. This function is documented on page 4.)

\DisableGenericHook Disabling a generic hook.

```

2786 \NewDocumentCommand \DisableGenericHook { m }
2787 { \hook_disable_generic:n {#1} }
```

(End of definition for \DisableGenericHook. This function is documented on page 4.)

```

2788 <latexrelease>\EndIncludeInRelease
2789 <latexrelease>\IncludeInRelease{2020/10/01}{\ActivateGenericHook}
2790 <latexrelease>          {Providing~hooks}
2791 <latexrelease>\def \ActivateGenericHook #1 { }
2792 <latexrelease>\def \DisableGenericHook #1 { }
2793 <latexrelease>\EndIncludeInRelease
```

\AddToHook

\AddToHookWithArguments

```
2794 <latexrelease>\IncludeInRelease{2023/06/01}{\AddToHookWithArguments}
2795 <latexrelease>{Hooks~with~args}
2796 \NewDocumentCommand \AddToHook { m o +m }
2797 { \hook_gput_code:nnn {#1} {#2} {#3} }
2798 \NewDocumentCommand \AddToHookWithArguments { m o +m }
2799 { \hook_gput_code_with_args:nnn {#1} {#2} {#3} }
2800 <latexrelease>\EndIncludeInRelease
2801 <latexrelease>\IncludeInRelease{2020/10/01}{\AddToHookWithArguments}
2802 <latexrelease>{Hooks~with~args}
2803 <latexrelease>\cs_new_protected:Npn \AddToHookWithArguments #1 #2 #3 { }
2804 <latexrelease>\EndIncludeInRelease
```

(End of definition for \AddToHook and \AddToHookWithArguments. These functions are documented on page 6.)

\AddToHookNext

\AddToHookNextWithArguments

```
2805 <latexrelease>\IncludeInRelease{2023/06/01}{\AddToHookNextWithArguments}
2806 <latexrelease>{Hooks~with~args}
2807 \NewDocumentCommand \AddToHookNext { m +m }
2808 { \hook_gput_next_code:nn {#1} {#2} }
2809 \NewDocumentCommand \AddToHookNextWithArguments { m +m }
2810 { \hook_gput_next_code_with_args:nn {#1} {#2} }
2811 <latexrelease>\EndIncludeInRelease
2812 <latexrelease>\IncludeInRelease{2020/10/01}{\AddToHookNextWithArguments}
2813 <latexrelease>{Hooks~with~args}
2814 <latexrelease>\cs_new_protected:Npn \AddToHookNextWithArguments #1 #2 { }
2815 <latexrelease>\EndIncludeInRelease
```

(End of definition for \AddToHookNext and \AddToHookNextWithArguments. These functions are documented on page 7.)

\ClearHookNext

```
2816 \NewDocumentCommand \ClearHookNext { m }
2817 { \hook_gclear_next_code:n {#1} }
```

(End of definition for \ClearHookNext. This function is documented on page 8.)

\RemoveFromHook

```
2818 \NewDocumentCommand \RemoveFromHook { m o }
2819 { \hook_gremove_code:nn {#1} {#2} }
```

(End of definition for \RemoveFromHook. This function is documented on page 6.)

\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.

```
2820 \NewDocumentCommand \SetDefaultHookLabel { m }
2821 { \__hook_set_default_hook_label:n {#1} }
```

The label is only automatically updated with `\@onefilewithoptions` (`\usepackage` and `\documentclass`), but some packages, like `TikZ`, define package-like interfaces, like `\usetikzlibrary` that are wrappers around `\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.

```

2822 \NewDocumentCommand \PushDefaultHookLabel { m }
2823 { \__hook_curr_name_push:n {#1} }
2824 \NewDocumentCommand \PopDefaultHookLabel { }
2825 { \__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`.

```

2826 \cs_gset_protected:Npn \@expl@push@filename@aux@@ #1#2#3
2827 {
2828   \__hook_curr_name_push:n {#3}
2829   \str_gset:Nx \g_file_curr_name_str {#3}
2830   #1 #2 {#3}
2831 }

```

(End of definition for `\SetDefaultHookLabel`, `\PushDefaultHookLabel`, and `\PopDefaultHookLabel`. These functions are documented on page 10.)

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

`\UseOneTimeHook`

`\UseHookWithArguments`

`\UseOneTimeHookWithArguments`

```

2832 <latexrelease>\IncludeInRelease{2023/06/01}{\UseHookWithArguments}
2833 <latexrelease> {Hooks~with~args}
2834 \cs_new:Npn \UseHook { \hook_use:n }
2835 \cs_new:Npn \UseOneTimeHook { \hook_use_once:n }
2836 \cs_new:Npn \UseHookWithArguments { \hook_use:nnw }
2837 \cs_new:Npn \UseOneTimeHookWithArguments { \hook_use_once:nnw }
2838 <latexrelease>\EndIncludeInRelease
2839 <latexrelease>\IncludeInRelease{2020/10/01}{\UseHookWithArguments}
2840 <latexrelease> {Hooks~with~args}
2841 <latexrelease>\cs_new:Npn \UseHookWithArguments #1 #2 { }
2842 <latexrelease>\cs_new:Npn \UseOneTimeHookWithArguments #1 #2 { }
2843 <latexrelease>\EndIncludeInRelease

```

(End of definition for `\UseHook` and others. These functions are documented on page 4.)

`\ShowHook`

`\LogHook`

```

2844 \cs_new_protected:Npn \ShowHook { \hook_show:n }
2845 \cs_new_protected:Npn \LogHook { \hook_log:n }

```

(End of definition for `\ShowHook` and `\LogHook`. These functions are documented on page 14.)

`\DebugHooksOn`

`\DebugHooksOff`

```

2846 \cs_new_protected:Npn \DebugHooksOn { \hook_debug_on: }
2847 \cs_new_protected:Npn \DebugHooksOff { \hook_debug_off: }

```

(End of definition for `\DebugHooksOn` and `\DebugHooksOff`. These functions are documented on page 15.)

`\DeclareHookRule`

```

2848 \NewDocumentCommand \DeclareHookRule { m m m m }
2849 { \hook_gset_rule:nnnn {#1}{#2}{#3}{#4} }

```

(End of definition for `\DeclareHookRule`. This function is documented on page 12.)

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

```
2850 \NewDocumentCommand \DeclareDefaultHookRule { m m m }
2851 { \hook_gset_rule:nnnn {??}{#1}{#2}{#3} }
2852 \@onlypreamble\DeclareDefaultHookRule
```

(End of definition for `\DeclareDefaultHookRule`. This function is documented on page 13.)

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

FMi: Needs perhaps an L3 interface, or maybe it should be dropped?

```
2853 \NewDocumentCommand \ClearHookRule { m m m }
2854 { \hook_gset_rule:nnnn {#1}{#2}{unrelated}{#3} }
```

(End of definition for `\ClearHookRule`. This function is documented on page 12.)

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

```
\IfHookEmptyT
\IfHookEmptyF
2855 \cs_new_eq:NN \IfHookEmptyTF \hook_if_empty:nTF
2856 \cs_new_eq:NN \IfHookEmptyT \hook_if_empty:nT
2857 \cs_new_eq:NN \IfHookEmptyF \hook_if_empty:nF
```

(End of definition for `\IfHookEmptyTF`, `\IfHookEmptyT`, and `\IfHookEmptyF`. These functions are documented on page 14.)

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

PhO: \IfHookExistsTF is used in `jlreq.cls`, `pxatbegshi.sty`, `pxeveryssel.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.

```
2858 \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` Deprecated.

```
\hook_provide:n
\hook_provide_reversed:n
\hook_provide_pair:nn
\_hook_activate_generic_reversed:n
\_hook_activate_generic_pair:nn
2859 \cs_new_protected:Npn \hook_disable:n
2860 {
2861   \_hook_deprecated_warn:nn
2862   { \hook_disable:n }
2863   { \hook_disable_generic:n }
2864   \hook_disable_generic:n
2865 }
2866 \cs_new_protected:Npn \hook_provide:n
2867 {
2868   \_hook_deprecated_warn:nn
2869   { \hook_provide:n }
2870   { \hook_activate_generic:n }
2871   \hook_activate_generic:n
2872 }
2873 \cs_new_protected:Npn \hook_provide_reversed:n
```

```

2874 {
2875     \__hook_deprecated_warn:nn
2876     { hook_provide_reversed:n }
2877     { hook_activate_generic:n }
2878     \__hook_activate_generic_reversed:n
2879 }
2880 \cs_new_protected:Npn \hook_provide_pair:nn
2881 {
2882     \__hook_deprecated_warn:nn
2883     { hook_provide_pair:nn }
2884     { hook_activate_generic:n }
2885     \__hook_activate_generic_pair:nn
2886 }
2887 \cs_new_protected:Npn \__hook_activate_generic_reversed:n #1
2888 { \__hook_normalize_hook_args:Nn \__hook_activate_generic:nn {#1} { - } }
2889 \cs_new_protected:Npn \__hook_activate_generic_pair:nn #1#2
2890 { \hook_activate_generic:n {#1} \__hook_activate_generic_reversed:n {#2} }

```

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

```

\DisableHook      Deprecated.
\ProvideHook      2891 \cs_new_protected:Npn \DisableHook
\ProvideReversedHook 2892 {
\ProvideMirroredHookPair 2893     \__hook_deprecated_warn:nn
2894     { DisableHook }
2895     { DisableGenericHook }
2896     \hook_disable_generic:n
2897 }
2898 \cs_new_protected:Npn \ProvideHook
2899 {
2900     \__hook_deprecated_warn:nn
2901     { ProvideHook }
2902     { ActivateGenericHook }
2903     \hook_activate_generic:n
2904 }
2905 \cs_new_protected:Npn \ProvideReversedHook
2906 {
2907     \__hook_deprecated_warn:nn
2908     { ProvideReversedHook }
2909     { ActivateGenericHook }
2910     \__hook_activate_generic_reversed:n
2911 }
2912 \cs_new_protected:Npn \ProvideMirroredHookPair
2913 {
2914     \__hook_deprecated_warn:nn
2915     { ProvideMirroredHookPair }
2916     { ActivateGenericHook }
2917     \__hook_activate_generic_pair:nn
2918 }

```

(End of definition for \DisableHook and others.)

```

\__hook_deprecated_warn:nn Warns about a deprecation, telling what should be used instead.
2919 \cs_new_protected:Npn \__hook_deprecated_warn:nn #1 #2
2920 { \msg_warning:nnnn { hooks } { deprecated } {#1} {#2} }

```

```

2921 \msg_new:nnn { hooks } { deprecated }
2922 {
2923   Command~\iow_char:N\|#1~is~deprecated~and~will~be~removed~in~a~
2924   future~release. \ \ \
2925   Use~\iow_char:N\|#2~instead.
2926 }

```

(End of definition for `\_hook_deprecated_warn:nn`.)

4.14 Internal commands needed elsewhere

Here we set up a few horrible (but consistent) L^AT_EX_{2 ϵ} 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.

```

2927 \@@=

```

```

\@expl@@@initialize@all@@
\@expl@@@hook@curr@name@pop@@
2928 \cs_new_eq:NN \@expl@@@initialize@all@@
2929 \_hook_initialize_all:
2930 \cs_new_eq:NN \@expl@@@hook@curr@name@pop@@
2931 \_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.

```

2932 %
2933 \<latexrelease>\IncludeInRelease{0000/00/00}{lthooks}
2934 \<latexrelease>{The~hook~management}%
2935 \<latexrelease>
2936 \<latexrelease>\def \NewHook#1{}
2937 \<latexrelease>\def \NewReversedHook#1{}
2938 \<latexrelease>\def \NewMirroredHookPair#1#2{}
2939 \<latexrelease>
2940 \<latexrelease>\def \DisableGenericHook #1{}
2941 \<latexrelease>
2942 \<latexrelease>\long\def\AddToHookNext#1#2{}
2943 \<latexrelease>
2944 \<latexrelease>\def\AddToHook#1{\@gobble@AddToHook@args}
2945 \<latexrelease>\providecommand\@gobble@AddToHook@args[2] [] {}
2946 \<latexrelease>
2947 \<latexrelease>\def\RemoveFromHook#1{\@gobble@RemoveFromHook@arg}
2948 \<latexrelease>\providecommand\@gobble@RemoveFromHook@arg[1] [] {}
2949 \<latexrelease>
2950 \<latexrelease>\def \UseHook #1{}
2951 \<latexrelease>\def \UseOneTimeHook #1{}
2952 \<latexrelease>\def \ShowHook #1{}
2953 \<latexrelease>\let \DebugHooksOn \@empty
2954 \<latexrelease>\let \DebugHooksOff \@empty
2955 \<latexrelease>
2956 \<latexrelease>\def \DeclareHookRule #1#2#3#4{}
2957 \<latexrelease>\def \DeclareDefaultHookRule #1#2#3{}
2958 \<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.

```

2959 <latexrelease>\long\def \IfHookExistsTF #1#2#3{#3}
2960 <latexrelease>\long\def \IfHookEmptyTF #1#2#3{#2}
2961 <latexrelease>
2962 <latexrelease>\EndModuleRelease
2963 <@@=hook>

2964 <latexrelease>\cs:w __hook_rollback_tidyng: \cs_end:
2965 <latexrelease>\bool_lazy_and:nnT
2966 <latexrelease>    { \int_compare_p:nNn { \sourceLaTeXdate } > { 20230600 } }
2967 <latexrelease>    { \int_compare_p:nNn { \requestedLaTeXdate } < { 20230601 } }
2968 <latexrelease> {
2969 <latexrelease>   \cs_gset_protected:Npn \__hook_rollback_tidyng:
2970 <latexrelease>   {
2971 <latexrelease>     \@latex@error { Rollback~code~executed~twice }
2972 <latexrelease>     {
2973 <latexrelease>       Something~went~wrong~(unless~this~was~
2974 <latexrelease>       done~on~purpose~in~a~testing~environment).
2975 <latexrelease>     }
2976 <latexrelease>     \use_none:nnnn
2977 <latexrelease>   }
2978 <latexrelease> \cs_set:Npn \__hook_tmp:w #1 #2
2979 <latexrelease> {
2980 <latexrelease>   \__hook_tl_gset:ce { __hook#1~#2 }
2981 <latexrelease>   {
2982 <latexrelease>     \exp_args:No \exp_not:o
2983 <latexrelease>     {
2984 <latexrelease>       \cs:w __hook#1~#2 \exp_last_unbraced:Ne \cs_end:
2985 <latexrelease>       { \__hook_braced_cs_parameter:n
2986 <latexrelease>         { __hook#1~#2 } }
2987 <latexrelease>     }
2988 <latexrelease>   }
2989 <latexrelease> }
2990 <latexrelease> \seq_map_inline:Nn \g__hook_all_seq
2991 <latexrelease> {
2992 <latexrelease>   \exp_after:wN \cs_gset_nopar:Npn
2993 <latexrelease>   \cs:w g__hook_#1_code_prop \exp_args:NNo \exp_args:No
2994 <latexrelease>   \cs_end: { \cs:w g__hook_#1_code_prop \cs_end: }
2995 <latexrelease>   \__hook_tmp:w { _toplevel } {#1}
2996 <latexrelease>   \__hook_tmp:w { _next } {#1}
2997 <latexrelease> }
2998 <latexrelease> }
2999 \ExplSyntaxOff
3000 </2ekernel | latexrelease>
3001 <@@=)

```

Index

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

Symbols	
<code>\</code>	1799, 1810, 2628, 2642, 2643, 2644, 2645, 2646, 2651, 2654, 2655, 2656, 2666, 2667, 2681, 2702, 2706, 2707, 2709, 2715, 2726, 2727, 2732, 2737, 2742, 2758, 2923, 2924, 2925
<code>\<addto-cmd></code>	5
<code>_</code>	2707
<code>_hook\textvisiblespace_meta_hook}</code>	34
<code>_hook_next\textvisiblespace_meta_hook}</code>	34
<code>_hook_toplevel\textvisiblespace_meta_hook}</code>	34
A	
<code>A/foo (hook)</code>	13, 13
<code>\ActivateGenericHook</code>	2782, 2784, 2789, 2791, 21
<code>\AddToHook</code>	2794, 2944, 36
<code>\AddToHookNext</code>	2805, 2942, 8
<code>\AddToHookNextWithArguments</code>	2805, 8
<code>\AddToHookWithArguments</code>	2794, 5
<code>after (hook)</code>	27
<code>\AfterEndEnvironment</code>	27
<code>\AtBeginDocument</code>	25
<code>\AtBeginEnvironment</code>	27
<code>\AtEndDocument</code>	29
<code>\AtEndEnvironment</code>	27
<code>\AtEndPreamble</code>	28
B	
<code>babel/<language>/afterextras (hook)</code>	22
<code>\BeforeBeginEnvironment</code>	27
<code>\begin</code>	3
<code>begindocument (hook)</code>	25, 25, 25, 28, 36
<code>begindocument/before (hook)</code>	28
<code>begindocument/end (hook)</code>	28
<code>\bfdefault</code>	30
<code>\bfseries (hook)</code>	30, 30
<code>\bfseries</code>	30
<code>\bfseries/defaults (hook)</code>	30, 30
bool commands:	
<code>\bool_gset_false:N</code>	15
<code>\bool_gset_true:N</code>	10
<code>\bool_if:NTF</code>	1945, 1954, 2045, 2054, 21, 334
<code>\bool_lazy_and:nnTF</code>	1947, 2047, 2497, 2965, 269
<code>\bool_lazy_and_p:nn</code>	2500
<code>\bool_lazy_or:nnTF</code>	908, 1898
<code>\bool_new:N</code>	6, 24
<code>\bool_set_false:N</code>	1937, 2037
<code>\bool_set_true:N</code>	1934, 2034
<code>\bool_while_do:nn</code>	1614, 1699
<code>\c_false_bool</code> 430, 478, 391, 397, 405, 52	
<code>\c_true_bool</code>	398, 406, 408
C	
<code>class (hook)</code>	22
<code>\ClearHookNext</code>	2816, 8
<code>\ClearHookRule</code>	2853, 2958, 12
<code>\clearpage</code>	29
clist commands:	
<code>\clist_gclear:N</code>	1613, 1698
<code>\clist_gput_left:Nn</code>	1517, 1552
<code>\clist_gput_right:Nn</code>	1519, 1556
<code>\clist_if_empty:NTF</code>	1964, 2065
<code>\clist_if_exist:NTF</code>	115
<code>\clist_map_inline:nn</code>	939, 948
<code>\clist_new:N</code>	117, 134
<code>\clist_use:Nn</code>	1966, 2067
<code>cmd (hook)</code>	55, 60, 16, 22, 22, 101
<code>cmd/<name>/after (hook)</code>	27
<code>cmd/<name>/before (hook)</code>	27
cs commands:	
<code>\cs:w</code>	1088, 1145, 1296, 1436, 1543, 1544, 1605, 1625, 1628, 1689, 1714, 1717, 1735, 1736, 1760, 1761, 1762, 1769, 1776, 2254, 2273, 2279, 2293, 2303, 2325, 2379, 2466, 2531, 2581, 2964, 2984, 2993, 2994, 375, 45
<code>\cs_end:</code>	789, 1088, 1145, 1299, 1439, 1543, 1544, 1605, 1625, 1628, 1689, 1714, 1717, 1735, 1736, 1751, 1761, 1762, 1769, 1776, 2131, 2253, 2257, 2261, 2273, 2278, 2279, 2293, 2299, 2313, 2326, 2331, 2379, 2465, 2466, 2531, 2581, 2964, 2984, 2994, 375
<code>\cs_generate_variant:Nn</code>	1102, 1125, 1655, 1738, 37, 47, 48, 51, 60
<code>\cs_gset:Npe</code>	1069
<code>\cs_gset:Npn</code>	1101, 1121, 1651, 2092, 2336, 2406

\cs_gset_eq:NN	792, 1446, 1472, 1487, 1488, 1491, 2242, 2343, 2344, 2345, 2346, 2610
\cs_gset_nopar:Npe	44, 46
\cs_gset_nopar:Npn	2992
\cs_gset_protected:Npe	1183, 20
\cs_gset_protected:Npn	603, 606, 624, 650, 699, 707, 1470, 1531, 2187, 2190, 2218, 2223, 2241, 2400, 2430, 2826, 2969, 82, 85, 94, 127, 154, 182, 185, 191, 215, 220, 245, 304, 322
\cs_if_exist:NTF	1073, 1138, 1845, 2512, 2541, 2549
\cs_if_exist_p:N	271
\cs_if_exist_use:NTF	1305, 1349, 1374
\cs_new:Npe	565
\cs_new:Npn	573, 926, 928, 930, 1224, 1236, 1241, 1249, 1258, 1260, 1267, 1294, 1301, 1303, 1418, 1434, 1568, 1569, 1740, 1741, 2088, 2095, 2251, 2276, 2284, 2297, 2305, 2314, 2323, 2358, 2539, 2547, 2561, 2579, 2588, 2593, 2834, 2835, 2836, 2837, 2841, 2842, 39, 40, 324, 330, 345, 355, 356, 358, 372, 378
\cs_new_eq:NN	1208, 1391, 1398, 1441, 1815, 1816, 1817, 1818, 1819, 1820, 2313, 2855, 2856, 2857, 2858, 2928, 2930, 7, 23, 34, 57
\cs_new_protected:Npn	425, 433, 449, 457, 467, 482, 493, 499, 505, 521, 534, 577, 594, 654, 673, 681, 688, 720, 726, 732, 742, 784, 786, 794, 796, 799, 801, 982, 984, 1019, 1056, 1068, 1071, 1100, 1113, 1115, 1117, 1119, 1136, 1160, 1167, 1325, 1332, 1363, 1385, 1392, 1399, 1405, 1410, 1415, 1416, 1444, 1497, 1577, 1659, 1742, 1749, 1758, 1765, 1772, 1779, 1787, 1793, 1804, 1821, 1828, 1840, 1847, 1854, 1860, 1862, 1866, 1984, 2106, 2124, 2129, 2138, 2156, 2170, 2192, 2205, 2232, 2236, 2246, 2259, 2268, 2282, 2288, 2311, 2318, 2341, 2364, 2374, 2385, 2391, 2411, 2441, 2443, 2453, 2595, 2601, 2607, 2778, 2779, 2780, 2803, 2814, 2844, 2845, 2846, 2847, 2859, 2866, 2873, 2880, 2887, 2889, 2891, 2898, 2905, 2912, 2919, 8, 13, 18, 41, 43, 45, 49, 52, 58, 63, 65, 67, 98, 142, 166, 168, 170, 195, 197, 199, 225, 260, 262, 279, 284, 286, 379, 388, 393, 401
\cs_parameter_spec:N	1252
\cs_replacement_spec	63
\cs_replacement_spec:N	1227, 1462, 1484, 1908, 1921, 1930
\cs_set:Npn	1169, 1180, 1187, 1196, 1905, 2599, 2605, 2978
\cs_set_eq:NN	1516, 1517, 1518, 1519, 1549, 1551, 1553, 1555, 1842, 1849, 1856
\cs_set_nopar:Npe	42
\cs_set_protected:Npn	414, 2110, 2141
\cs_to_str:N	45
\cs_undefine:N	671, 1106, 1107, 1108, 1109, 1129, 1130, 1131, 1132, 1156, 1212, 1232, 1290, 1311, 1312, 1323, 1417, 2351, 2437, 2575, 73, 74, 84, 126, 184, 190, 265
\csname	8
D	
debug commands:	
\debug_resume:	33
\debug_suspend:	33
\DebugHooksOff	2846, 2954, 15
\DebugHooksOn	2846, 2953, 15
\DeclareDefaultHookRule	2850, 2957, 13
\DeclareHookRule	2848, 2956, 7
\def	2791, 2792, 2936, 2937, 2938, 2940, 2942, 2944, 2947, 2950, 2951, 2952, 2956, 2957, 2958, 2959, 2960, 109
\DisableGenericHook	2786, 2792, 2940, 36
\DisableHook	2891, 4
\DiscardShipoutBox	8
\document	28
\documentclass	107
E	
else commands:	
\else:	789, 1063, 1220, 1282, 1423, 1430, 1438, 1754, 2300, 2328, 2470, 2486, 2533, 2568, 2583
\end	26
\endcsname	8
enddocument (hook)	24, 25, 29
\enddocument	28
enddocument/afteraux (hook)	29
enddocument/afterlastpage (hook)	29
enddocument/end (hook)	29
enddocument/info (hook)	29
\EndIncludeInRelease	600, 651, 668, 672, 695, 716, 750, 808, 838, 870, 897, 900, 920, 923, 936, 954, 959, 962, 971, 976, 979, 1016, 1053, 1103, 1110, 1126,

1133, 1153, 1157, 1209, 1213, 1229, 1233, 1261, 1264, 1287, 1291, 1308, 1313, 1320, 1324, 1360, 1384, 1467, 1494, 1528, 1567, 1656, 1739, 1980, 2084, 2089, 2094, 2184, 2191, 2215, 2231, 2238, 2243, 2265, 2285, 2315, 2333, 2338, 2348, 2352, 2355, 2382, 2397, 2408, 2426, 2438, 2450, 2460, 2490, 2509, 2572, 2576, 2658, 2661, 2775, 2781, 2788, 2793, 2800, 2804, 2811, 2815, 2838, 2843, 79, 95, 123, 139, 151, 163, 179, 192, 212, 222, 242, 257, 275, 281, 301, 319, 323	\exp_not:n 554, 557, 1070, 1094, 1193, 1197, 1203, 1204, 1652, 2982, 108, 235 \exp_stop_f: 1272, 1420, 1428 expand@font@defaults (hook) 30 \expanded 45 \ExplSyntaxOff 2999 \ExplSyntaxOn 3
\EndModuleRelease 2962 env (hook) 13, 22 env/(env)/after (hook) 26, 26, 27 env/(env)/before (hook) 26, 26, 27 env/(env)/begin (hook) 26, 27 env/(env)/end (hook) 26, 27 env/document/after (hook) 30 env/document/before (hook) 28 env/document/begin (hook) 28 env/document/end (hook) 30 env/quote/after (hook) 20, 20 \ERROR 1740, 1741 \errorstopmode 18	F fi commands: \fi: 791, 1065, 1222, 1245, 1285, 1424, 1432, 1438, 1756, 2135, 2255, 2263, 2280, 2302, 2330, 2467, 2472, 2488, 2535, 2570, 2585 file (hook) 13, 22 file commands: \g_file_curr_name_str 2829 file/before (hook) 101 foo (hook) 99
exp commands: \exp:w 779, 833, 55 \exp_after:wN . . 420, 567, 778, 832, 1062, 1064, 1205, 1370, 1543, 1753, 1872, 1905, 1990, 2134, 2279, 2463, 2466, 2530, 2992, 50, 55, 374, 375, 55 \exp_args:Nc 1226, 1239, 2220 \exp_args:NcV 591 \exp_args:Ne 427, 475, 551, 736, 738, 2368, 2369 \exp_args:NNe 1524, 1645 \exp_args:Nne 1882 \exp_args:Nnnv 251 \exp_args:NNo . . 1086, 1143, 1734, 2993 \exp_args:NNV 1618, 1705 \exp_args:NNx 1562 \exp_args:No 1086, 1143, 1204, 2093, 2152, 2982, 2993 \exp_args:Nv . . 1271, 2022, 2030, 2132 \exp_end: 779, 833 \exp_last_unbraced:Ne 1088, 1121, 1145, 1238, 2984 \exp_last_unbraced:Nf . . . 1251, 1929 \exp_last_unbraced:NNf 1179 \exp_last_unbraced:NNNNo . . 2087, 377 \exp_not:N 567, 568, 569, 573, 574, 1173, 1177, 1196, 1199, 1200, 1202, 1246, 1508, 1510, 1646, 1648, 385	G \g__hook_\meta_\{hook}_code_prop 34 \g__hook_\meta_\{hook}_declared_tl . . 34 \g__hook_\meta_\{hook}_parameter_tl . . 34 \g__hook_\meta_\{hook}_reversed_tl . . 34 group commands: \group_begin: 1172, 1185, 381 \group_end: 1176, 1195, 384
	H hook commands: \hook_activate_generic:n 2785, 2871, 2890, 2903, 282, 282, 284, 302, 320, 322, 42 \hook_debug_off: 2847, 7, 13, 18 \hook_debug_on: 2846, 7, 8, 18 \hook_disable:n 2859, 2859 \hook_disable_generic:n . . . 2787, 2864, 2896, 258, 258, 260, 276, 279, 16 \hook_gclear_next_code:n 2232, 2232, 2817, 17 \hook_gput_code:nnn 491, 491, 493, 601, 603, 703, 724, 2797, 17 \hook_gput_code_with_args:nnn 491, 499, 650, 2799, 17 \hook_gput_next_code:nn . 712, 730, 2154, 2154, 2156, 2185, 2187, 2808, 53 \hook_gput_next_code_with_ args:nn 2170, 2190, 2810, 17 \hook_gremove_code:nn 980, 980, 982, 1017, 2819, 17 \hook_gset_rule:nnnn 1325, 1325, 2849, 2851, 2854, 18 \hook_if_empty:n 2474, 2476, 2491, 2493

```

\hook_if_empty:nTF .....
    1894, 2000, 2474, 2855, 2856, 2857, 8
\hook_if_empty_p:n 1949, 2049, 2474, 18
\hook_log:n ..... 1840, 1840, 2845, 18
\hook_new:n 883, 2762, 61, 63, 82, 217, 42
\hook_new_pair:nn .....
    ..... 2766, 193, 195, 215, 15
\hook_new_pair_with_args:nn .... 16
\hook_new_pair_with_args:nnn ...
    .... 2774, 193, 193, 197, 213, 220, 16
\hook_new_reversed:n .....
    ..... 2764, 164, 166, 182, 218, 15
\hook_new_reversed_with_args:nn .
    .... 2772, 164, 164, 168, 180, 191, 16
\hook_new_with_args:nn .....
    ..... 2770, 61, 61, 65, 80, 94, 16
\hook_provide:n ..... 2859, 2866
\hook_provide_pair:nn ... 2859, 2880
\hook_provide_reversed:n . 2859, 2873
\hook_show:n 1840, 1847, 1854, 2844, 84
\hook_use:n 1487, 2244, 2244, 2246,
    2266, 2268, 2286, 2288, 2343, 2834, 97
\hook_use:nnw ... 2284, 2314, 2316,
    2316, 2318, 2334, 2336, 2344, 2836, 16
\hook_use_once:n .....
    ..... 2383, 2385, 2400, 2835, 16
\hook_use_once:nnw .....
    2383, 2383, 2391, 2398, 2406, 2837, 16
hook internal commands:
\c_hook_ ..... 1317, 1321
\g_hook_??_code_prop ..... 1314
\c_hook_??_parameter_tl ..... 1314
\g_hook_??_reversed_tl ..... 1314
\g_hook_{hook}_code_prop ..... 34
\g_hook_{hook}_labels_clist .... 38
\g_hook_{hook}_reversed_tl .... 34
\__hook_activate_generic:n .... 282
\__hook_activate_generic:nn ....
    ..... 2888, 285, 286, 304
\__hook_activate_generic_pair:nn
    ..... 2859, 2885, 2889, 2917
\__hook_activate_generic_-
    reversed:n .....
    ..... 2859, 2878, 2887, 2890, 2910
\g_hook_all_seq .....
    ..... 1448, 1475, 2990, 28, 102, 131
\__hook_apply_rule_>:nnn ... 1815
\__hook_apply_rule_<:nnn ... 1815
\__hook_apply_rule_<:nnn .... 1815
\__hook_apply_rule_>:nnn .... 1815
\__hook_apply_rule_xE:nnn ... 1815
\__hook_apply_rule_xW:nnn ... 1815
\__hook_apply_label_pair:nnn ...
    1595, 1596, 1677, 1678, 1742, 1742, 78
\__hook_apply_rule:nnn .....
    ..... 1752, 1758, 1758, 81
\__hook_apply_rule:nnnN ..... 82
\__hook_apply_rule_>:nnn .... 1793
\__hook_apply_rule_<:nnn .... 1793
\__hook_apply_rule_<:nnn .... 1765
\__hook_apply_rule_>:nnn .... 1765
\__hook_apply_rule_xE:nnn .... 1779
\__hook_apply_rule_xW:nnn .... 1779
\__hook_braced_cs_parameter:n ...
    ..... 1089, 1146, 1171,
    1182, 1234, 1234, 1236, 1262, 2985
\__hook_braced_hidden_loop:w ...
    ..... 1234, 1238, 1241, 1247
\__hook_braced_parameter:n .....
    ..... 1265, 1265, 1267,
    1288, 1290, 1509, 1511, 1646, 1648
\__hook_braced_real_loop:w ... 1265
\__hook_chk_args_allowed:nn ....
    507, 652, 652, 654, 669, 671, 2208, 92
\__hook_clear_next:n . 2212, 2228,
    2233, 2234, 2234, 2236, 2239, 2241, 89
\__hook_clist_gput:Nn 1517, 1519,
    1551, 1555, 1619, 1706, 1740, 1741
\__hook_code_gset:nn . 1111, 1111,
    1113, 1125, 1127, 1129, 1506, 2445, 112
\__hook_code_gset_aux:nnn . 1074,
    1111, 1114, 1116, 1118, 1119, 1132
\__hook_code_gset_auxi:nnnn ....
    1054, 1075, 1100, 1102, 1109, 1140, 62
\__hook_cs_end:w .....
    ..... 1234, 1253, 1254, 1255, 1260
\__hook_cs_gput_right:nnn .....
    ..... 546, 1054,
    1054, 1056, 1104, 1106, 2213, 2416
\__hook_cs_gput_right_fast:nnn ..
    ..... 1054, 1062, 1068, 1107
\__hook_cs_gput_right_slow:nnn ..
    ..... 1054, 1064, 1071, 1108
\__hook_cs_if_empty:N .....
    ..... 1214, 1216, 1230, 1232
\__hook_cs_if_empty:NTF ... 1214,
    1919, 1926, 2209, 2211, 2482, 2483
\__hook_cs_if_empty_p:N ..... 1214
\__hook_cs_parameter_count:N ...
    ..... 1234, 1239, 1249
\__hook_cs_parameter_count:w ...
    ..... 1234, 1251, 1258, 1259
\l_hook_cur_hook_tl .....
    ..... 1581, 1663, 1799, 1810, 29, 83
\__hook_curr_name_pop: .....
    ..... 449, 2825, 2931, 411, 47
\__hook_curr_name_push:n .....
    ..... 425, 2823, 2828, 411, 46

```

__hook_curr_name_push_aux:n . . .	\c__hook_generic_class/./before_-
. 427 , 433 , 411	tl 937
__hook_currname_or_default: . . .	\c__hook_generic_cmd/./after_tl 937
. 543 , 633 , 2673 ,	\c__hook_generic_cmd/./before_tl 937
2721 , 327 , 337 , 341 , 357 , 358 , 44	\c__hook_generic_env/./after_tl 937
__hook_debug:n	\c__hook_generic_env/./before_tl 937
. 536 , 626 , 1447 , 1456 , 1474 , 1477 ,	\c__hook_generic_env/./begin_tl 937
1499 , 1523 , 1533 , 1561 , 1600 , 1620 ,	\c__hook_generic_env/./end_tl . . 937
1682 , 1707 , 1767 , 1774 , 1781 , 1789 ,	\c__hook_generic_file/./after_tl 937
1795 , 1806 , 2160 , 2174 , 7 , 7 , 20 , 31	\c__hook_generic_file/./before_-
\g__hook_debug_bool 6 , 10 , 15 , 21	tl 937
__hook_debug_gset: 7 , 11 , 16 , 18	\c__hook_generic_include/./after_-
__hook_debug_label_data:N	tl 937
. . 1600 , 1642 , 1683 , 1731 , 1828 , 1828	\c__hook_generic_include/./before_-
__hook_debug_print_rules:n	tl 937
. 2138 , 2138	\c__hook_generic_include/./end_-
__hook_declare_deprecated_-	tl 937
generic:NNn 778 , 799 , 832	\c__hook_generic_package/./after_-
__hook_declare_deprecated_-	tl 937
generic:NNw 794 , 800 , 801	\c__hook_generic_package/./before_-
__hook_deprecated_generic_-	tl 937
warn:n 777 , 784 , 831 ,	__hook_generic_parameter:n
1009 , 1045 , 1336 , 1367 , 1870 , 1988 , 55	 1081 , 1301 , 1312
__hook_deprecated_generic_-	__hook_generic_parameter:w
warn:Nn 784	 1302 , 1303
__hook_deprecated_generic_-	\c__hook_generics_file_prop 913 , 960
warn:Nw 784	\c__hook_generics_prop
__hook_deprecated_generic_-	 849 , 881 , 937 , 955 , 957
warn:w 785 , 786	\c__hook_generics_reversed_ii_-
__hook_deprecated_warn:nn	prop 858 , 885 , 960
. 2861 , 2868 , 2875 , 2882 ,	\c__hook_generics_reversed_iii_-
2893 , 2900 , 2907 , 2914 , 2919 , 2919	prop 862 , 889 , 960
__hook_disable:n 258 , 261 , 262	__hook_gput_code:nnn
__hook_do_deprecated_generic:Nn	 491 , 496 , 502 , 505 , 605 , 606 , 684
. 794 ,	__hook_gput_code_store:nnn
794 , 1010 , 1046 , 1337 , 1368 , 1871 , 1989	 491 , 518 , 521
__hook_do_deprecated_generic:Nw	__hook_gput_next_code:nn
. 794 , 795 , 796	 691 , 2159 , 2173 , 2189 , 2192 , 2192
__hook_double_hashes:n	__hook_gput_next_do:nn
. 558 , 1086 , 1095 , 1143 , 1200 , 67	 692 , 713 , 730 ,
\c__hook_empty_tl 1306 , 35	2198 , 2203 , 2203 , 2205 , 2216 , 2218 , 53
__hook_end_document_label_-	__hook_gput_next_do:Nnn . 2220 , 2223
check: 456 , 457 , 464 , 411	__hook_gput_undeclared_hook:nnn
__hook_file_hook_normalize:n . . .	 673 , 673 , 685 , 704 , 724 , 53
. 738 , 921 , 921 , 924 , 926 , 2369 , 58	__hook_gremove_code:nn
\l__hook_front_tl	 980 , 983 , 984 , 1011 , 1019 , 1047
. 1570 , 1611 , 1614 , 1617 , 1619 ,	__hook_gset_rule:nnnn 1325 , 1327 ,
1620 , 1621 , 1635 , 1636 , 1696 , 1700 ,	1330 , 1332 , 1337 , 1361 , 1363 , 1368
1704 , 1706 , 1708 , 1710 , 1724 , 1725	__hook_hash_check:nTF . 491 , 556 , 565
\c__hook_generic_⟨type⟩/./⟨place⟩_tl	__hook_hash_check_aux:w 491 , 567 , 573
. 55	\c__hook_hash_tl 1246 , 67
\c__hook_generic_class/./after_-	\c__hook_hashes_tl 1173 , 66
tl 937	

```

\g__hook_hook_curr_name_tl .....
..... 411, 423, 444, 445,
452, 462, 463, 489, 32, 360, 370, 47
\__hook_hook_gput_code_do:nnn ...
491, 525, 534, 613, 624, 676, 234, 251
\__hook_if_cmd_hook:n 2557, 2559, 2573
\__hook_if_cmd_hook:nTF .....
..... 2100, 2557, 2575
\__hook_if_cmd_hook:w ... 2560, 2561
\__hook_if_cmd_hook:wTF ..... 2557
\__hook_if_cmd_hook_p:n ..... 2557
\__hook_if_cmd_hook_p:w ..... 2557
\__hook_if_declared:n ..... 2522
\__hook_if_declared:nTF .....
. 658, 883, 1059, 1077, 2099, 2522,
69, 87, 172, 201, 204, 291, 309, 35
\__hook_if_declared_p:n ..... 2522
\__hook_if_deprecated_generic:n 2545
\__hook_if_deprecated_generic:nTF
..... 775, 829, 1007,
1043, 1334, 1365, 1868, 1986, 2537
\__hook_if_deprecated_generic:w .
..... 2546, 2547
\__hook_if_deprecated_generic-
p:n ..... 2537
\__hook_if_disabled:n ..... 267
\__hook_if_disabled:nTF .....
..... 529, 617, 1880, 1892, 1975,
1998, 2078, 2194, 258, 288, 306, 36
\__hook_if_disabled_p:n ..... 258
\__hook_if_execute_immediately:n
..... 2461
\__hook_if_execute_immediately:nTF
508, 608, 1341, 2387, 2393, 2402, 2461
\__hook_if_execute_immediately-
p:n ..... 2461
\__hook_if_file_hook:w . 898, 901, 903
\__hook_if_file_hook:wTF .....
..... 734, 898, 2366, 54
\__hook_if_file_hook_p:w ..... 898
\__hook_if_generic:n ..... 2537
\__hook_if_generic:nTF .....
..... 758, 816, 1080, 1878, 1994, 2537
\__hook_if_generic:w ... 2538, 2539
\__hook_if_generic_p:n ..... 2537
\__hook_if_generic_reversed:n . 2577
\__hook_if_generic_reversed:nTF .
..... 770, 824, 2577, 296, 314
\__hook_if_generic_reversed:w ...
..... 2578, 2579
\__hook_if_generic_reversed_p:n 2577
\__hook_if_label_case:nnnn .....
..... 1434, 1434, 1593, 1675, 2115
\__hook_if_replacing_args: ... 2591
\__hook_if_replacing_args:TF ...
.. 510, 553, 580, 656, 1093, 2587,
2593, 2598, 2599, 2604, 2605, 2610
\__hook_if_reversed:n ..... 2528
\__hook_if_reversed:nTF .....
..... 1515, 1548, 1915,
1955, 1957, 2016, 2055, 2058, 2528
\__hook_if_reversed_p:n ..... 2528
\__hook_if_structure_exist:n . 2516
\__hook_if_structure_exist:nTF ..
..... 986, 1021,
2197, 2495, 2516, 2752, 144, 156, 35
\__hook_if_structure_exist_p:n 2516
\__hook_if_usable:n ..... 2510
\__hook_if_usable:nTF .....
.... 523, 537, 582, 611, 627, 760,
818, 851, 1003, 1039, 1502, 1537,
1890, 1914, 1973, 1996, 2014, 2076,
2161, 2175, 2419, 2510, 2858, 100, 60
\__hook_if_usable_p:n 1948, 2048, 2510
\__hook_if_usable_use:n .....
..... 2353, 2368, 2371, 2374, 96
\__hook_include_legacy_code-
chunk:n ..... 1501,
1536, 120, 136, 223, 223, 225, 243, 245
\__hook_init_structure:n .....
.. 545, 635, 675, 1347, 1372, 2207,
113, 133, 140, 140, 142, 152, 154, 50
\__hook_initialize_all: ... 1442,
1442, 1444, 1468, 1470, 1492, 2929
\__hook_initialize_hook_code:n ..
..... 1446, 1473, 1495, 1495,
1497, 1529, 1531, 2262, 2283, 2312, 81
\__hook_initialize_single:NNn ...
..... 1521, 1559,
1575, 1575, 1577, 1655, 1657, 1659, 73
\l__hook_label_0_tl ..... 1570
\__hook_label_if_exist_apply:nnnTF
..... 1742, 1744, 1746, 1749, 81
\__hook_label_ordered:nn ... 1426, 73
\__hook_label_ordered:nnTF .....
..... 1389, 1396, 1403, 1426, 71
\__hook_label_ordered_p:nn ... 1426
\__hook_label_pair:nn .....
..... 1388, 1395, 1402, 1407,
1412, 1417, 1418, 1418, 1824, 1825, 72
\l__hook_labels_int .....
..... 1570, 1580, 1584,
1616, 1638, 1662, 1666, 1702, 1727, 79
\l__hook_labels_seq .. 1570, 1579,
1585, 1603, 1661, 1667, 1686, 1830
\__hook_list_if_rule_exists:nnnTF
..... 2106, 2126, 2127, 2129

```

```

\__hook_list_one_rule:nnn .....
..... 2106, 2117, 2118, 2124
\__hook_list_rules:nn .....
..... 1935, 2035, 2106, 2106, 2143, 90
\__hook_log:nN .....
..... 1840, 1843, 1850, 1857,
1864, 1866, 1871, 1982, 1984, 1989
\__hook_log_cmd:n ..... 1842,
1849, 1856, 1861, 1863, 1875, 1993
\__hook_log_line:n .....
1840, 1860, 1891, 1893, 1897, 1911,
1923, 1933, 1951, 1970, 1997, 1999,
2003, 2011, 2025, 2033, 2051, 2072
\__hook_log_line_indent:n .....
..... 1840, 1862,
1901, 1907, 1917, 1924, 1938, 1946,
2005, 2008, 2018, 2026, 2038, 2046
\__hook_log_next_code:n .....
..... 2030, 2085, 2085, 2090, 2092
\__hook_log_next_code:w .. 1929, 2088
\__hook_make_name:n .....
..... 351, 357, 366, 372, 372, 45
\__hook_make_name:w ... 372, 374, 378
\__hook_make_usable:n .....
..... 822, 855, 91, 127, 312
\__hook_make_usable:nn ..... 765,
768, 75, 96, 96, 98, 124, 126, 294, 38
\__hook_misused_if_replacing_
args:nn ..... 2587, 2588, 2594
\__hook_msg_pair_found:nnn .....
..... 1767, 1774,
1781, 1789, 1797, 1808, 1821, 1821
\g__hook_name_stack_seq .... 412,
413, 417, 424, 444, 451, 459, 469, 32
\__hook_new:n ..... 83, 85, 187
\__hook_new:nn .....
..... 61, 64, 66, 67, 84, 175, 207
\__hook_new_pair:nnn .. 196, 198, 199
\__hook_new_reversed:n ..... 183, 185
\__hook_new_reversed:nn .....
..... 164, 167, 169, 170, 184, 190, 208
\__hook_next_⟦hook⟧ ..... 60
\__hook_next_gset:nn ..... 992,
1111, 1117, 1131, 2212, 2237, 2446, 148
\c__hook_nine_parameters_tl ....
..... 1082, 1197, 1906, 35, 108
\__hook_normalize_code_pool:n ...
..... 1158, 1158, 1160, 1210, 1212, 114, 38
\__hook_normalize_cs_args:nn 1134,
1134, 1136, 1154, 1156, 110, 111, 38
\__hook_normalize_fn:nn .....
..... 586, 1164, 1183, 1208, 65
\__hook_normalize_hook_args:Nn ..
1843, 1850, 1857, 2159, 2173, 2188,
2233, 2388, 2394, 2403, 2888, 64,
66, 83, 167, 169, 183, 261, 285, 379, 388
\__hook_normalize_hook_args:Nnn .
..... 496, 502, 604, 983, 196, 198, 379, 393
\__hook_normalize_hook_args_-
aux:Nn ..... 379, 379, 390, 395, 403
\__hook_normalize_hook_rule_-
args:Nnnnn ..... 1327, 379, 401
\__hook_parameter:n .. 1122, 1180,
1292, 1292, 1294, 1309, 1311, 1885
\c__hook_parameter_cmd ..... 972, 977
\c__hook_parameter_cmd/./after_-
tl ..... 972
\c__hook_parameter_cmd/./before_-
tl ..... 972
\__hook_parse_dot_label:nN .....
..... 430, 478, 328, 330, 330
\__hook_parse_dot_label:w .....
..... 330, 342, 345
\__hook_parse_dot_label_aux:w ...
..... 330, 348, 356
\__hook_parse_dot_label_cleanup:w
..... 330, 352, 355
\__hook_parse_label_default:nN ..
..... 324, 324, 391, 397, 398, 405, 406, 408
\__hook_patch_cmd_or_delay:Nnn .. 55
\__hook_post_initialization_-
defs: ..... 1465,
2339, 2339, 2341, 2346, 2349, 2351
\__hook_preamble_hook:n .....
..... 1488, 1874, 1992,
2244, 2248, 2259, 2272, 2282, 2292,
2311, 2320, 2345, 2378, 2413, 2432, 96
\__hook_print_args:n ..... 2095
\__hook_print_args:nn ... 1882, 2095
\__hook_prop_gput_labeled_-
cleanup:nnn ..... 491, 551, 577
\__hook_prop_gput_labeled_do:Nnn
..... 591, 594
\__hook_prop_gput_labeled_-
do:Nnnnn ..... 491
\l__hook_rear_tl .....
... 1570, 1601, 1607, 1608, 1631,
1632, 1684, 1692, 1693, 1720, 1721
\__hook_replacement_spec:N 1218, 1224
\__hook_replacing_args_false: ...
..... 495, 663,
1451, 2158, 2415, 2587, 2601, 231, 48
\__hook_replacing_args_reset: ...
..... 497, 503,
1454, 2168, 2182, 2418, 2587, 2607, 237
\__hook_replacing_args_true: ...
..... 501, 1452, 2172, 2587, 2595
\g__hook_replacing_stack_seq . 2587

```

```

\l__hook_return_tl .....
.... 451, 452, 459, 463, 579, 587,
592, 596, 597, 642, 645, 999, 1034,
1617, 1618, 1704, 1705, 2609, 2610, 25
\__hook_rollback_tidyng: .... 2969
\__hook_rule<_gset:nnn ..... 1385
\__hook_rule>_gset:nnn ..... 1385
\__hook_rule_after_gset:nnn ....
..... 1385, 1392, 1398
\__hook_rule_before_gset:nnn ...
..... 1385, 1385, 1391, 78
\__hook_rule_gclear:nnn .....
..... 1348, 1373, 1415, 1416, 72
\__hook_rule_incompatible-error-
gset:nnn ..... 1405
\__hook_rule_incompatible-warning-
gset:nnn ..... 1405
\__hook_rule_unrelated_gset:nnn .
..... 1415, 1415, 72
\__hook_rule_voids_gset:nnn ....
..... 1399, 1399
\__hook_seq_csname:n .....
..... 1568, 1569, 1587,
1621, 1669, 1710, 1770, 1777, 1835
\__hook_set_default_hook_label:n
..... 467, 467, 2821
\__hook_set_default_label:n ....
..... 467, 475, 482
\__hook_set_normalize_fn:nn ....
..... 584, 1158, 1162, 1167, 65
\__hook_str_compare:nn .....
..... 1420, 1428, 1437, 23, 23
\__hook_strip_double_slash:n ...
..... 921, 927, 928
\__hook_strip_double_slash:w ...
..... 921, 929, 930, 934
\__hook_tl_csname:n .....
..... 1568, 1568, 1574, 1586,
1602, 1605, 1607, 1611, 1623, 1625,
1628, 1631, 1636, 1668, 1685, 1689,
1691, 1696, 1712, 1714, 1717, 1719,
1725, 1768, 1769, 1775, 1776, 1834
\__hook_tl_gclear:N .....
... 1026, 1027, 1031, 1612, 1697,
2455, 2456, 2457, 58, 58, 60, 238, 253
\__hook_tl_gput:Nn .....
..... 1516, 1518, 1549, 1553,
1618, 1645, 1705, 1734, 1740, 1740, 79
\__hook_tl_gput_left:Nn .....
..... 1516, 1550, 52, 52
\__hook_tl_gput_right:Nn .....
..... 636, 1518, 1554,
1647, 1736, 1738, 2229, 49, 49, 51
\__hook_tl_gset:Nn ..... 1387,
1394, 1401, 1406, 1411, 1541, 2228,
2442, 2980, 43, 43, 45, 47, 48, 50, 54
\__hook_tl_gset_eq:NN ..... 57, 57, 59
\__hook_tl_set:Nn .....
..... 1173, 1586, 1668, 41, 41, 33
\__hook_tmp:w .....
... 414, 418, 420, 1169, 1180, 1196,
1202, 1205, 1905, 1908, 2110, 2132,
2141, 2152, 2978, 2995, 2996, 34, 34
\l__hook_tmpa_bool 1934, 1937, 1945,
1954, 2034, 2037, 2045, 2054, 24, 86
\l__hook_tmpa_tl .. 424, 1181, 1203, 25
\l__hook_tmpb_tl . 1170, 1177, 1205, 25
\__hook_toplevel_⟨hook⟩ ..... 60
\__hook_toplevel_gset:nn .....
991, 996, 1111, 1115, 1130, 2447, 147
\__hook_try_declaring_generic_-
hook:nnn ..... 531, 619,
678, 679, 681, 697, 699, 718, 721, 53
\__hook_try_declaring_generic_-
hook:nNNnn .. 723, 729, 732, 732, 54
\__hook_try_declaring_generic_-
hook:wn ..... 752,
755, 810, 813, 840, 843, 872, 875
\__hook_try_declaring_generic_-
hook:wnTF ..... 683,
690, 701, 710, 745, 751, 804, 55
\__hook_try_declaring_generic_-
hook_split:nNNnn 732, 737, 740, 743
\__hook_try_declaring_generic_-
next_hook:nn .....
..... 678, 688, 708, 727, 2199, 53
\__hook_try_file_hook:n .....
..... 2353, 2361, 2364, 96
\__hook_try_put_cmd_hook:n .....
..... 764, 821, 854
\__hook_update_hook_code:n .....
..... 526, 614, 1004, 1040, 1352,
1377, 1441, 1441, 1446, 1453, 1472,
1476, 2210, 2226, 76, 297, 315, 73
\__hook_use:wn .....
2295, 2309, 2353, 2353, 2356, 2358, 96
\__hook_use_end: .... 2303, 2306, 2313
\__hook_use_i_delimit_by_s_-
mark:nw ..... 1244, 39, 40
\__hook_use_initialized:n .....
..... 1487, 2244, 2249,
2251, 2276, 2297, 2343, 2420, 2434, 93
\__hook_use_initialized:nnw ....
..... 2316, 2321, 2323, 2344
\__hook_use_none_delimit_by_s_-
mark:w 1339, 1345, 2463, 2530, 39, 39
\__hook_use_once:n ..... 2403, 2430

```


\prg_return_true:	772, 826, 865, 892, 914, 1219, 1429, 2469, 2485, 2504, 2507, 2513, 2519, 2525, 2532, 2542, 2553, 2567, 2582, 272
prop commands:	
\prop_clear:N	1163
\prop_const_from_keyval:Nn	957, 965, 967, 969
\prop_gclear:N	990, 1025, 1447, 1474
\prop_gclear_new:N	2448, 2458
\prop_get:NnN	587, 1617, 1703
\prop_get:NnNTF	596, 641
\prop_gpop:NnNTF	998, 1033
\prop_gput:Nnn	597, 598, 644, 647, 1524, 1562, 2612, 2613, 2614
\prop_gset_eq:NN	1165
\prop_if_empty:NTF	1504, 1539, 2004, 2481
\prop_if_empty_p:N	1900, 2498
\prop_if_exist:NTF	2108, 2480, 2518
\prop_if_exist_p:N	1899
\prop_if_in:NnTF	849, 857, 861, 881, 884, 888, 913
\prop_map_break:	1594, 1676, 2116
\prop_map_function:NN	1164, 65
\prop_map_inline:Nn	1459, 1480, 1582, 1589, 1591, 1664, 1671, 1673, 1832, 1903, 2007, 2111, 2113
\prop_new:N	1314, 1315, 30, 31, 146, 158
\prop_put:Nnn	1199, 1802, 1813
\prop_set_eq:NN	1520, 1557
\prop_show:N	90
\providecommand	2945, 2948
\ProvideHook	2891
\ProvideMirroredHookPair	2891
\ProvideReversedHook	2891
\PushDefaultHookLabel	2820, 10
Q	
quark commands:	
\quark_if_recursion_tail_stop:n	416
\q_recursion_stop	422
\q_recursion_tail	421, 422
\q_stop	569, 574
R	
\RemoveFromHook	2818, 2947, 6
\requestedLaTeXdate	2967
\RequirePackage	10
rmfamily (hook)	30, 30, 30
\rmfamily	30
S	
scan commands:	
\scan_new:N	38
\scan_stop:	683, 690, 701, 710, 745, 756, 792, 804, 814, 844, 876, 1218, 2578, 2579
scan internal commands:	
\s_hook_mark	734, 785, 787, 795, 797, 800, 802, 904, 929, 930, 934, 1239, 1256, 1260, 1302, 1303, 1358, 2295, 2309, 2358, 2366, 2469, 2471, 2532, 2534, 2538, 2539, 2546, 2547, 2560, 2561, 38, 39, 40, 342, 345, 348, 352, 355, 356, 33
selectfont (hook)	31
\selectfont	30
seq commands:	
\seq_clear:N	1579, 1661
\seq_clear_new:N	1587, 1669
\seq_gpop:NN	2609
\seq_gpop:NNTF	451, 459
\seq_gpop_right:NN	424
\seq_gpush:Nn	444, 2597, 2603
\seq_gput_right:Nn	413, 417, 102, 131
\seq_if_empty:NTF	412, 469
\seq_map_inline:Nn	1448, 1475, 1603, 1621, 1686, 1709, 2990
\seq_new:N	1570, 2587, 28, 33
\seq_put_right:Nn	1585, 1667, 1770, 1777
\seq_use:Nnnn	1830, 1835
\SetDefaultHookLabel	2820, 8
sffamily (hook)	30
\sffamily	30
\shipout	30
shipout/lastpage (hook)	29
show commands:	
\show_hook:n	84
\ShowHook	2844, 2952, 14
\small	7
\sourceLaTeXdate	2966
\space	629
\special	29
str commands:	
\c_hash_str	568, 574
\c_right_brace_str	2088
\str_count:n	1271, 1885, 37, 37
\str_gset:Nn	2829
\str_if_eq:nn	73
\str_if_eq:nnTF	438, 484, 541, 543, 630, 632, 762, 820, 853, 906, 988, 995, 1023, 1030, 1823, 1941, 2041, 2148, 2360, 2564, 2618, 2688, 340
\str_if_eq_p:nn	910, 1614, 1700
str internal commands:	
__str_if_eq:nn	23, 32
\strut	31

T	
test (hook)	22, 23
TeX and L ^A T _E X 2 _ε commands:	
\@...hook	41
\@begindocumenthook	25
\@cls@pkg	2721
\@currname	423, 362, 368, 108
\@currnamestack	420, 108
\@empty	2953, 2954
\@expl@@@hook@curr@name@pop@@	2928
\@expl@@@initialize@call@@	1491, 2928
\@expl@push@filename@aux@@	2826, 108
\@firstofone	5
\@gobble@AddToHook@args	2944, 2945
\@gobble@RemoveFromHook@arg	2947, 2948
\@kernel@after@hook	24
\@kernel@after@enddocument@afterlastpage	455
\@kernel@before@hook	24
\@latex@error	2971
\@onefilewithoptions	107
\@onlypreamble	2852
\@pushfilename	108
\@spaces	539, 1863, 1963, 1973, 2064, 2075, 2164, 2178
\@@end	29
\expand@font@defaults	30
\g@addto@macro	38
\on@line	538, 629, 1500, 1535, 2163, 2177
\protected	93
tex commands:	
\tex_escapechar:D	1186, 66
tl commands:	
\c_empty_tl	2242, 59
\c_novalue_tl	44
\tl_const:Nn	941, 942, 944, 945, 946, 947, 950, 951, 953, 974, 975, 1319, 35, 36, 104, 107
\tl_gclear:N	92
\tl_gclear_new:N	264
\tl_gput_right:Nn	455, 75
\tl_gset:Nn	411, 423, 445, 489, 771, 825, 859, 863, 886, 890, 176, 188, 295, 313
\tl_gset_eq:NN	452, 463, 57
\tl_if_blank:nTF	429, 435, 477
\tl_if_empty:N	99
\tl_if_empty:NTF	659, 1060, 1450, 2020, 2028, 2225, 2227, 229, 249, 360, 362
\tl_if_empty:nTF	576, 846, 878, 932, 2551, 332, 347, 350
\tl_if_empty_p:N	2501, 2502
\tl_if_empty_p:n	909
\tl_if_exist:N	93
\tl_if_exist:NTF	1297, 2270, 2290, 2376, 2524, 2565, 129, 227, 247
\tl_if_exist_p:N	270
\tl_if_novalue:nTF	326
\tl_log:n	1843
\tl_new:N	1316, 1572, 1573, 1574, 25, 26, 27, 29, 32, 72, 90, 118, 132, 135, 159, 160, 293, 311
\tl_set:Nn	579, 1170, 1177, 1181, 1581, 1601, 1602, 1607, 1608, 1623, 1630, 1632, 1663, 1684, 1685, 1691, 1693, 1712, 1719, 1721, 1768, 1775
\tl_set_eq:NN	1611, 1635, 1696, 1724
\tl_show:n	1850, 1857, 1961, 2062
\tl_to_str:n	540, 568, 629, 2009, 2022, 2093, 2165, 2179, 378
\tl_trim_spaces:n	407
\tl_trim_spaces_apply:nN	328
\tl_use:N	1078, 1141, 1834
token commands:	
\token_if_macro:NTF	1226, 1252
\token_to_str:N	375
top-level (hook)	25, 6, 6
ttfamily (hook)	30
\ttfamily	30
U	
use commands:	
\use:N	1651, 1784, 1785, 1825, 2337, 2407, 2423
\use:n	516, 571, 1174, 1649, 2102, 2136, 2389, 2395, 2404, 232, 382
\use:nn	1755, 2152, 2727
\use_i:nn	2254, 2599
\use_i:nnn	2101, 73
\use_ii:nn	659, 2605, 73
\use_iii:nn	73
\use_none:n	660, 1370, 1441, 1488, 1753, 2256, 2345, 7, 93
\use_none:nn	2093, 2134
\use_none:nnn	766, 2099
\use_none:nnnn	2976
\use_none:nnnnn	1872, 1990
\use_none:nnnnnnnn	1370, 1872, 1990
\UseHook	2832, 2950, 4
\UseHookWithArguments	2832, 22
\UseOneTimeHook	2832, 2951, 5
\UseOneTimeHookWithArguments	2832, 5
\usepackage	27
\usetikzlibrary	10
W	
\write	29