

L^AT_EX's hook management*

Frank Mittelbach[†]

September 13, 2023

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	10
2.1.8	Querying hooks	11
2.1.9	Displaying hook code	12
2.1.10	Debugging hook code	13
2.2	L3 programming layer (<code>expl3</code>) interfaces	14
2.3	On the order of hook code execution	17
2.4	The use of “reversed” hooks	18
2.5	Difference between “normal” and “one-time” hooks	19
2.6	Generic hooks provided by packages	20
2.7	Hooks with arguments	21
2.8	Private L ^A T _E X kernel hooks	22
2.9	Legacy L ^A T _E X 2 _ε interfaces	23
3	L^AT_EX 2_ε commands and environments augmented by hooks	23
3.1	Generic hooks	24
3.1.1	Generic hooks for all environments	24
3.1.2	Generic hooks for commands	25
3.1.3	Generic hooks provided by file loading operations	25
3.2	Hooks provided by <code>\begin{document}</code>	26
3.3	Hooks provided by <code>\end{document}</code>	26
3.4	Hooks provided by <code>\shipout</code> operations	28
3.5	Hooks provided for paragraphs	28
3.6	Hooks provided in NFSS commands	28
3.7	Hook provided by the mark mechanism	29

*This module has version v1.1f dated 2023/08/11, © L^AT_EX Project.

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

4	The Implementation	29
4.1	Debugging	29
4.2	Borrowing from internals of other kernel modules	30
4.3	Declarations	30
4.4	Providing new hooks	32
4.4.1	The data structures of a hook	32
4.4.2	On the existence of hooks	33
4.4.3	Setting hooks up	34
4.4.4	Disabling and providing hooks	40
4.5	Parsing a label	42
4.6	Adding or removing hook code	46
4.7	Setting rules for hooks code	66
4.8	Specifying code for next invocation	87
4.9	Using the hook	88
4.10	Querying a hook	94
4.11	Messages	98
4.12	L ^A T _E X 2 _ε package interface commands	101
4.13	Deprecated that needs cleanup at some point	104
4.14	Internal commands needed elsewhere	106
	Index	108

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 `exp13` 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 hooks run when loading files (see section 3.1).

<u><code>\NewHook</code></u>	<code>\NewHook {<hook>}</code>
------------------------------	--------------------------------------

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.

<u><code>\NewReversedHook</code></u>	<code>\NewReversedHook {<hook>}</code>
--------------------------------------	--

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.

<u><code>\NewMirroredHookPair</code></u>	<code>\NewMirroredHookPair {<hook-1>} {<hook-2>}</code>
--	---

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.

<u><code>\NewHookWithArguments</code></u>	<code>\NewHookWithArguments {<hook>} {<number>}</code>
---	--

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.

<u><code>\NewReversedHookWithArguments</code></u>	<code>\NewReversedHookWithArguments {<hook>} {<number>}</code>
---	--

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.

<u><code>\NewMirroredHookPairWithArguments</code></u>	<code>\NewMirroredHookPairWithArguments {<hook-1>} {<hook-2>} {<number>}</code>
---	---

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.

<code>\DisableGenericHook</code>	<code>\DisableGenericHook {⟨hook⟩}</code>
----------------------------------	---

After this declaration¹ the `⟨hook⟩` is no longer usable: Any further attempt to add code to it will result in an error and any use, e.g., via `\UseHook`, will simply do nothing.

This is intended to be used with generic command hooks (see `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 `⟨hook⟩` can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

<code>\ActivateGenericHook</code>	<code>\ActivateGenericHook {⟨hook⟩}</code>
-----------------------------------	--

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

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

2.1.3 Using hooks in code

<code>\UseHook</code>	<code>\UseHook {⟨hook⟩}</code>
-----------------------	--------------------------------

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

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

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

<code>\UseHookWithArguments</code>	<code>\UseHookWithArguments {⟨hook⟩} {⟨number⟩} {⟨arg₁⟩} ... {⟨arg_n⟩}</code>
------------------------------------	--

Execute the code stored in the `⟨hook⟩` and pass the arguments `{⟨arg1⟩}` through `{⟨argn⟩}` to the `⟨hook⟩`. Otherwise, it works exactly like `\UseHook`. The `⟨number⟩` should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove `⟨number⟩` items from the input. Section 2.7 explains hooks with arguments.

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

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.

`\UseOneTimeHook` `\UseOneTimeHook {<hook>}`

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

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

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

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

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

`\UseOneTimeHookWithArguments` `\UseOneTimeHookWithArguments {<hook>} {<number>} {<arg1>} ... {<argn>}`

Works exactly like `\UseOneTimeHook`, but passes arguments `{<arg1>}` through `{<argn>}` to the `<hook>`. The `<number>` should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove `<number>` items from the input.

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

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

2.1.4 Updating code for hooks

`\AddToHook` `\AddToHook {<hook>}[<label>]{<code>}`

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

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

The hook doesn’t have to exist for code to be added to it. However, if it is not declared, then obviously the added `<code>` will never be executed. This allows for hooks to work regardless of package loading order and enables packages to add to hooks from other packages without worrying whether they are actually used in the current document. See section 2.1.8.

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

\AddToHookWithArguments	\AddToHookWithArguments {<hook>}[<label>]{<code>}
--------------------------------	--

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note that you can't cancel the setting with

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

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

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

`\AddToHookNext` `\AddToHookNext {<hook>}{<code>}`

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

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

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

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

`\AddToHookNext{<hook>}{<code-1>\AddToHookNext{<hook>}{<code-2>}}`

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

A hook doesn’t have to exist for code to be added to it. This allows for hooks to work regardless of package loading order. See section 2.1.8.

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

`\AddToHookNextWithArguments` `\AddToHookNextWithArguments {<hook>}{<code>}`

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

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

`\ClearHookNext` `\ClearHookNext{<hook>}`

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

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

2.1.5 Hook names and default labels

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

Using an explicit $\langle label \rangle$ 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 $\langle label \rangle$ 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 $\langle hook \rangle$ and $\langle label \rangle$ 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 $\langle hook \rangle$ or $\langle label \rangle$ contains a non-expandable non-character token, a low-level `TeX` error is raised (namely, the $\langle hook \rangle$ is expanded using `TeX`'s `\csname...\endcsname`, as such, Unicode characters are allowed in $\langle hook \rangle$ and $\langle label \rangle$ 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 $\langle labels \rangle$ 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 $\langle hook \rangle$ and in a $\langle label \rangle$. If the $\langle hook \rangle$ name or the $\langle label \rangle$ consist just of a single dot (`.`), or starts with a dot followed by a slash (`./`) then the dot denotes the $\langle default label \rangle$ (usually the current package or class name—see `\SetDefaultHookLabel`). A `“.”` or `“./”` anywhere else in a $\langle hook \rangle$ or in $\langle label \rangle$ 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 $\langle default label \rangle$ 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 $\langle default label \rangle$. This may have exceptions—see `\PushDefaultHookLabel`.

This syntax is available in all $\langle label \rangle$ arguments and most $\langle hook \rangle$ arguments, both in the `LATEX 2ε` interface, and the `LATEX 3` interface described in section 2.2.

Important:

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

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

<code>\PushDefaultHookLabel</code>	<code>\PushDefaultHookLabel {$\langle default label \rangle$}</code>
<code>\PopDefaultHookLabel</code>	<code>$\langle code \rangle$</code>

`\PopDefaultHookLabel`

`\PushDefaultHookLabel` sets the current $\langle default label \rangle$ to be used in $\langle label \rangle$ arguments, or when replacing a leading “.” (see above). `\PopDefaultHookLabel` reverts the $\langle default label \rangle$ to its previous value.

Inside a package or class, the $\langle default label \rangle$ is equal to the package or class name, unless explicitly changed. Everywhere else, the $\langle default label \rangle$ 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{ $\langle package name \rangle$ }
 $\langle package code \rangle$ 
\PopDefaultHookLabel
```

to set the $\langle default label \rangle$ for the package or class file. Inside the $\langle package code \rangle$ the $\langle default label \rangle$ 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 $\langle label \rangle$! (that is, the $\langle default label \rangle$ 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 $\langle default label \rangle$ 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 precaution to determine if some other package got loaded as well (before or after) and 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 code added to hooks got added in the right order and some cases even altering the loading order wouldn't resolve the conflicts.

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

<code>\DeclareHookRule</code>	<code>\DeclareHookRule {<hook>}{<label1>}{<relation>}{<label2>}</code>
-------------------------------	--

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

Currently, the supported relations are the following:

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

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

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

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

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

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

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

<code>\ClearHookRule</code>	<code>\ClearHookRule {<hook>}{<label1>}{<label2>}</code>
-----------------------------	--

Syntactic sugar for saying that `<label1>` and `<label2>` are unrelated for the given `<hook>`.

<code>\DeclareDefaultHookRule</code>	<code>\DeclareDefaultHookRule {<label1>}{<relation>}{<label2>}</code>
--------------------------------------	---

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

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

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

2.1.8 Querying hooks

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

- exist and be empty;

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

- 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.

<code>\IfHookEmptyTF</code>	<code>\IfHookEmptyTF</code>
★	<code>{\langle hook \rangle}{\langle true code \rangle}{\langle false code \rangle}</code>

Tests if the `\langle hook \rangle` is empty (*i.e.*, no code was added to it using either `\AddToHook` or `\AddToHookNext`) or such code was removed again (via `\RemoveFromHook`), and branches to either `\langle true code \rangle` or `\langle false code \rangle` depending on the result.

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

2.1.9 Displaying hook code

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

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

Displays information about the `\langle hook \rangle` 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 `\langle hook \rangle` 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

$\langle label \rangle_{\square} \rightarrow_{\square} \langle code \rangle$

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

Document-level $_{\square}$ (top-level) $_{\square}$ code $_{\square}$ (executed $_{\square}$ $\langle first|last \rangle$):
 $\square\square\square\square \rightarrow_{\square} \langle top-level\ code \rangle$

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

$\rightarrow_{\square} \langle next-code \rangle$

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{<label>}[example-hook]`.

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

$\langle label-1 \rangle | \langle label-2 \rangle_{\square} \text{with}_{\square} \langle default? \rangle_{\square} \text{relation}_{\square} \langle relation \rangle$

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

<code>\DebugHooksOn</code>	<code>\DebugHooksOn</code>
<code>\DebugHooksOff</code>	

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

2.2 L3 programming layer (expl3) interfaces

This is a quick summary of the L^AT_EX3 programming interfaces for use with packages written in `expl3`. In contrast to the L^AT_EX 2_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.

<code>\hook_new:n</code>	<code>\hook_new:n {\langle hook \rangle}</code>
<code>\hook_new_reversed:n</code>	<code>\hook_new_reversed:n {\langle hook \rangle}</code>
<code>\hook_new_pair:nn</code>	<code>\hook_new_pair:nn {\langle hook-1 \rangle} {\langle hook-2 \rangle}</code>

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 {<hook>}</code>
<code>\hook_use:nnw</code>	<code>\hook_use:nnw {<hook>} {<number>} {<arg₁>} ... {<arg_n>}</code>

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

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

<code>\hook_use_once:n</code>	<code>\hook_use_once:n {<hook>}</code>
<code>\hook_use_once:nnw</code>	<code>\hook_use_once:nnw {<hook>} {<number>} {<arg₁>} ... {<arg_n>}</code>

Changes the `{<hook>}` status so that from now on any addition to the hook code is executed immediately. Then execute any `{<hook>}` code already set up. `\hook_use_once:nnw` should be used for hooks declared with arguments, and should be followed by as many brace groups as the declared number of arguments. The `<number>` should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove `<number>` items from the input.

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

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

Adds a chunk of `<code>` to the `<hook>` labeled `<label>`. If the label already exists the `<code>` is appended to the already existing code.

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

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

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

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

Adds a chunk of `<code>` for use only in the next invocation of the `<hook>`. Once used it is gone.

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

This is simpler than `\hook_gput_code:nnn`, the code is simply appended to the hook in the order of declaration at the very end, i.e., after all standard code for the hook got executed. Thus if one needs to undo what the standard does one has to do that as part of `<code>`.

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

<hr/> <code>\hook_gclear_next_code:n</code> <hr/>	<code>\hook_gclear_next_code:n {<hook>}</code> Undo any earlier <code>\hook_gput_next_code:nn</code> .
<hr/> <code>\hook_gremove_code:nn</code> <hr/>	<code>\hook_gremove_code:nn {<hook>} {<label>}</code> Removes any code for <code><hook></code> labeled <code><label></code> . If there is no code under the <code><label></code> in the <code><hook></code> , or if the <code><hook></code> does not exist, a warning is issued when you attempt to use <code>\hook_gremove_code:nn</code> , and the command is ignored. If the second argument is <code>*</code> , 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 <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.
<hr/> <code>\hook_gset_rule:nnnn</code> <hr/>	<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 <code>??</code> 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/> <code>\hook_if_empty_p:n</code> \star <code>\hook_if_empty:nTF</code> \star <hr/>	<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 <code>.</code> is treated literally.
<hr/> <code>\hook_show:n</code> <code>\hook_log:n</code> <hr/>	<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 <code>.log</code> file, and <code>\hook_show:n</code> prints them to the terminal/command window and starts <code>TeX</code> '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/> <code>\hook_debug_on:</code> <code>\hook_debug_off:</code> <hr/>	<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}

```

⁴there are simpler ways to achieve the same effect.

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

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

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

2.5 Difference between “normal” and “one-time” hooks

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

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

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

This has some consequences one needs to be aware of:

- If `<code>` is added to a normal hook after the hook was executed and it is never executed again for one or the other reason, then this new `<code>` will never be executed.
- In contrast if that happens with a one-time hook the `<code>` is executed immediately.

In particular this means that construct such as

```
\AddToHook{myhook}
{ <code-1> \AddToHook{myhook}{<code-2>}{<code-3>} }
```

works for one-time 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>`,

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

- 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.

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 TeX's limit of 9 arguments applies, and a low-level TeX error is raised if you try to reference an argument number that doesn't exist.

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

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

then, in the `<code>` of the hook, all instances of `#1` will be replaced by `arg-1`, `#2` by `arg-2` and so on. If, at the point of usage, the programmer provides more arguments than the hook is declared to take, the excess arguments are simply ignored by the hook. Behaviour is 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:

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.

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

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

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

Running the code above prints in the terminal:

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

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

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

2.8 Private \LaTeX kernel hooks

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

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

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

which means first the user/package-accessible `enddocument` hook is executed and then the internal kernel hook. As their name indicates these kernel commands should not be altered by third-party packages, so please refrain from that in the interest of stability and instead use the public hook next to it.⁷

⁷As with everything in \TeX there is not enforcement of this rule, and by looking at the code it is

2.9 Legacy L^AT_EX 2_ε interfaces

L^AT_EX 2_ε 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 L^AT_EX and more will follow. See the next section for what is already available.

<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 L^AT_EX 2_ε 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 L^AT_EX we will turn this legacy support off, as it does unnecessary slow down the processing.

3 L^AT_EX 2_ε commands and environments augmented by hooks

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

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!”

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 “ $\langle type \rangle / \langle name \rangle / \langle position \rangle$ ”, where $\langle type \rangle$ is from the predefined list shown below, and $\langle name \rangle$ is the variable part whose meaning will depend on the $\langle type \rangle$. The last component, $\langle position \rangle$, 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 L^AT_EX 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 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 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.

Generic environment hooks are never one-time hooks even with environments that are supposed to appear only once in a document.⁸ In contrast to other hooks there is also no need to declare them using `\NewHook`.

The hooks are only executed if `\begin{env}` and `\end{env}` is used. If the environment code is executed via low-level calls to `\env` and `\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>[\label] {env} {code}</code>	This declaration adds to the <code>env/⟨env⟩/before</code> hook using the <code>⟨label⟩</code> . If <code>⟨label⟩</code> is not given, the <code>⟨default label⟩</code> is used (see section 2.1.5).
<hr/> <hr/>	<code>\AtBeginEnvironment</code>	<code>[\label] {env} {code}</code>	This is like <code>\BeforeBeginEnvironment</code> but it adds to the <code>env/⟨env⟩/begin</code> hook.
<hr/> <hr/>	<code>\AtEndEnvironment</code>	<code>[\label] {env} {code}</code>	This is like <code>\BeforeBeginEnvironment</code> but it adds to the <code>env/⟨env⟩/end</code> hook.
<hr/> <hr/>	<code>\AfterEndEnvironment</code>	<code>[\label] {env} {code}</code>	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 L^AT_EX 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 L^AT_EX'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`.

⁸Thus if one adds code to such hooks after the environment has been processed, it will only be executed if the environment appears again and if that doesn't happen the code will never get executed.

3.2 Hooks provided by `\begin{document}`

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

Supported are now the following hooks (all of them one-time hooks):

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

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

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

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

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

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

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

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

3.3 Hooks provided by `\end{document}`

$\text{\LaTeX} 2_{\epsilon}$ 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 $\text{\LaTeX} 2.09$, it was not flexible enough for a number of use cases and so packages, such as `etoolbox`, `atveryend` and others patched `\enddocument` to add additional points where code could be hooked into.

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

Supported are now the following hooks (all of them one-time hooks):

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

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

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

enddocument/afterlastpage As the name indicates this hook should not receive code that generates material for further pages. It is the right place to do some final housekeeping and possibly write out some information to the `.aux` file (which is still open at this point to receive data, but since there will be no more pages you need to write to it using `\immediate\write`). It is also the correct place to set up any testing code to be run when the `.aux` file is re-read in the next step.

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

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

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

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

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

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

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

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

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

There is also the hook `shipout/lastpage`. This hook is executed as part of the last `\shipout` in the document to allow package to add final `\special`'s to that page. Where this hook is executed in relation to those from the above list can vary from document to document. Furthermore to determine correctly which of the `\shipouts` is the last one,

L^AT_EX 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 L^AT_EX has finished the document processing.

3.4 Hooks provided by `\shipout` operations

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

3.5 Hooks provided for paragraphs

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

3.6 Hooks provided in NFSS commands

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

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

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

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

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

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

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

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

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

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

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

3.7 Hook provided by the mark mechanism

See `ltmarks-doc.pdf` for details.

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

4 The Implementation

```

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:Npx \__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 16.)

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`.)

`\tl_gremove_once:Nx` Some variants of `expl3` functions.

`\tl_show:x` *FMi: should probably be moved to `expl3`*

`\tl_log:x`

`\tl_set:Ne`

`\cs_replacement_spec:c`

`\prop_put:Nne`

`\str_count:e`

```

37 \cs_generate_variant:Nn \tl_gremove_once:Nn { Nx }
38 \cs_generate_variant:Nn \tl_show:n { x }
39 \cs_generate_variant:Nn \tl_log:n { x }
40 \cs_generate_variant:Nn \tl_set:Nn { Ne }
41 \cs_generate_variant:Nn \cs_replacement_spec:N { c }
42 \cs_generate_variant:Nn \prop_put:Nnn { Nne }
43 \cs_generate_variant:Nn \str_count:n { e }

```

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

`\s__hook_mark` Scan mark used for delimited arguments.

`\scan_new:N \s__hook_mark`

(End of definition for `\s__hook_mark`.)

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

`__hook_use_i_delimit_by_s_mark:nw`

```

45 \cs_new:Npn \__hook_use_none_delimit_by_s_mark:w #1 \s__hook_mark { }
46 \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`.

```

47 \cs_new_protected:Npn \__hook_tl_set:cn #1#2
48 { \cs_set_nopar:cp {#1} { \__kernel_exp_not:w {#2} } }

```

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

`__hook_tl_gset:Nn` Same as above.

`__hook_tl_gset:Nx`

`__hook_tl_gset:cn`

`__hook_tl_gset:co`

`__hook_tl_gset:cx`

```

49 \cs_new_protected:Npn \__hook_tl_gset:Nn #1#2
50 { \cs_gset_nopar:Npx #1 { \__kernel_exp_not:w {#2} } }
51 \cs_new_protected:Npn \__hook_tl_gset:Nx #1#2
52 { \cs_gset_nopar:Npx #1 {#2} }
53 \cs_generate_variant:Nn \__hook_tl_gset:Nn { c, co }
54 \cs_generate_variant:Nn \__hook_tl_gset:Nx { 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`

```

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

```

(End of definition for `_hook_tl_gput_right:Nn`.)

`_hook_tl_gput_left:Nn` Same as above.

```

58 \cs_new_protected:Npn \_hook\_tl\_gput\_left:Nn #1#2
59 {
60   \_hook\_tl\_gset:Nx #1
61   { \_kernel\_exp\_not:w {#2} \_kernel\_exp\_not:w \exp\_after:wN {#1} }
62 }

```

(End of definition for `_hook_tl_gput_left:Nn`.)

`_hook_tl_gset_eq:NN` Same as above.

```

63 \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
64 \cs_new_protected:Npn \_hook\_tl\_gclear:N #1
65 { \_hook\_tl\_gset\_eq:NN #1 \c\_empty\_tl }
66 \cs_generate_variant:Nn \_hook\_tl\_gclear:N { c }

```

(End of definition for `_hook_tl_gclear:N`.)

4.4 Providing new hooks

4.4.1 The data structures of a hook

`\g_hook_code_prop` Hooks have a name (called $\langle hook \rangle$ in the description below) and for each hook we have to provide a number of data structures. These are

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

`\g_hook_declared_tl` `\g_hook_rule_label1|label2_tl` A token list holding the relation between $\langle label1 \rangle$ and $\langle label2 \rangle$ in the $\langle hook \rangle$. The $\langle labels \rangle$ are lexically (reverse) sorted to ensure that two labels always point to the same token list. For global rules, the $\langle hook \rangle$ name is ??.

`\g_hook_parameter_tl` `\g_hook_rule_label1|label2_tl` A token list holding the relation between $\langle label1 \rangle$ and $\langle label2 \rangle$ in the $\langle hook \rangle$. The $\langle labels \rangle$ are lexically (reverse) sorted to ensure that two labels always point to the same token list. For global rules, the $\langle hook \rangle$ name is ??.

`\g_hook_next_hook` `\g_hook_rule_label1|label2_tl` A token list holding the relation between $\langle label1 \rangle$ and $\langle label2 \rangle$ in the $\langle hook \rangle$. The $\langle labels \rangle$ are lexically (reverse) sorted to ensure that two labels always point to the same token list. For global rules, the $\langle hook \rangle$ name is ??.

`\g_hook_toplevel_hook` `\g_hook_rule_label1|label2_tl` A token list holding the relation between $\langle label1 \rangle$ and $\langle label2 \rangle$ in the $\langle hook \rangle$. The $\langle labels \rangle$ are lexically (reverse) sorted to ensure that two labels always point to the same token list. For global rules, the $\langle hook \rangle$ 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_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 $\langle reversed \rangle$ 1 is +1 for normal hooks and -1 for reversed ones.

`\g_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 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`.

```
67 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_new_with_args:nn}
68 <latexrelease>{Hooks~with~args}
69 \cs_new_protected:Npn \hook_new:n #1
70 { \__hook_normalize_hook_args:Nn \__hook_new:nn {#1} { 0 } }
71 \cs_new_protected:Npn \hook_new_with_args:nn #1 #2
72 { \__hook_normalize_hook_args:Nn \__hook_new:nn {#1} {#2} }
73 \cs_new_protected:Npn \__hook_new:nn #1 #2
74 {
```

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.

```

75   \__hook_if_declared:nTF {#1}
76   { \msg_error:nnn { hooks } { exists } {#1} }
77   {
78     \tl_new:c { g__hook_#1_declared_tl }
79     \cs_undefine:c { __hook~#1 }
80     \cs_undefine:c { c__hook_#1_parameter_tl }
81     \__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`).

```

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

```

(End of definition for `\hook_new:n`, `\hook_new_with_args:nn`, and `__hook_new:nn`. These functions are documented on page 14.)

`__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.

```

102 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_make_usable:nn}
103 <latexrelease> {Hooks~with~args}
104 \cs_new_protected:Npn \__hook_make_usable:nn #1 #2
105 {

```

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.

```

106   \__hook_if_usable:nF {#1}
107   {
108     \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`).

```

109      \__kernel_cs_parm_from_arg_count:nnF
110      { \tl_const:cn { c__hook_#1_parameter_tl } } {#2}
111      {
112          \msg_error:nnnn { hooks } { too-many-args } {#1} {#2}
113          \tl_const:cx { c__hook_#1_parameter_tl }
114          { \exp_not:V \c__hook_nine_parameters_tl }
115      }

```

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

```

116      \__hook_normalise_cs_args:nn { _toplevel } {#1}
117      \__hook_normalise_cs_args:nn { _next } {#1}

```

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

```

118      \__hook_code_gset:nn {#1} { }

```

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

```

119      \__hook_init_structure:n {#1}

```

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

```

120      \__hook_normalise_code_pool:n {#1}

```

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

```

121      \clist_if_exist:cF { g__hook_#1_labels_clist }
122      {
123          \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.

```

124      \tl_new:c { g__hook_#1_reversed_tl }
125      }

```

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!

```

126      \__hook_include_legacy_code_chunk:n {#1}
127      }
128  }
129  <latexrelease>\EndIncludeInRelease

```

```

130 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_make_usable:nn}
131 <latexrelease>{Hooks-with-args}
132 <latexrelease>\cs_undefine:N \__hook_make_usable:nn
133 <latexrelease>\cs_gset_protected:Npn \__hook_make_usable:n #1
134 <latexrelease> {
135 <latexrelease> \tl_if_exist:cF { \__hook~#1 }
136 <latexrelease> {
137 <latexrelease> \seq_gput_right:Nn \g__hook_all_seq {#1}
138 <latexrelease> \tl_new:c { \__hook~#1 }
139 <latexrelease> \__hook_init_structure:n {#1}
140 <latexrelease> \clist_new:c { g__hook_#1_labels_clist }
141 <latexrelease> \tl_new:c { g__hook_#1_reversed_tl }
142 <latexrelease> \__hook_include_legacy_code_chunk:n {#1}
143 <latexrelease> }
144 <latexrelease> }
145 <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.

```

146 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_init_structure:n}
147 <latexrelease>{Hooks-with-args}
148 <latexrelease>\cs_new_protected:Npn \__hook_init_structure:n #1
149 <latexrelease> {
150 <latexrelease> \__hook_if_structure_exist:nF {#1}
151 <latexrelease> {
152 <latexrelease> \prop_new:c { g__hook_#1_code_prop }
153 <latexrelease> \__hook_toplevel_gset:nn {#1} { }
154 <latexrelease> \__hook_next_gset:nn {#1} { }
155 <latexrelease> }
156 <latexrelease> }
157 <latexrelease>\EndIncludeInRelease

158 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_init_structure:n}
159 <latexrelease>{Hooks-with-args}
160 <latexrelease>\cs_gset_protected:Npn \__hook_init_structure:n #1
161 <latexrelease> {
162 <latexrelease> \__hook_if_structure_exist:nF {#1}
163 <latexrelease> {
164 <latexrelease> \prop_new:c { g__hook_#1_code_prop }
165 <latexrelease> \tl_new:c { \__hook_toplevel~#1 }
166 <latexrelease> \tl_new:c { \__hook_next~#1 }
167 <latexrelease> }
168 <latexrelease> }
169 <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
170 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_new_reversed_with_args:nn}
171 <latexrelease> {Hooks~with~args}
172 \cs_new_protected:Npn \hook_new_reversed:n #1
173 { \__hook_normalize_hook_args:Nn \__hook_new_reversed:nn {#1} { 0 } }
174 \cs_new_protected:Npn \hook_new_reversed_with_args:nn #1 #2
175 { \__hook_normalize_hook_args:Nn \__hook_new_reversed:nn {#1} {#2} }
176 \cs_new_protected:Npn \__hook_new_reversed:nn #1 #2
177 {
178   \__hook_if_declared:nTF {#1}
179   { \msg_error:nnn { hooks } { exists } {#1} }
180   {
181     \__hook_new:nn {#1} {#2}
182     \tl_gset:cn { g__hook_#1_reversed_tl } { - }
183   }
184 }
185 <latexrelease>\EndIncludeInRelease
186 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_new_reversed_with_args:nn}
187 <latexrelease> {Hooks~with~args}
188 <latexrelease>\cs_gset_protected:Npn \hook_new_reversed:n #1
189 <latexrelease> { \__hook_normalize_hook_args:Nn \__hook_new_reversed:n {#1} }
190 <latexrelease>\cs_undefine:N \__hook_new_reversed:nn
191 <latexrelease>\cs_gset_protected:Npn \__hook_new_reversed:n #1
192 <latexrelease> {
193 <latexrelease> \__hook_new:n {#1}
194 <latexrelease> \tl_gset:cn { g__hook_#1_reversed_tl } { - }
195 <latexrelease> }
196 <latexrelease>\cs_undefine:N \__hook_new_reversed:nn
197 <latexrelease>\cs_gset_protected:Npn \hook_new_reversed_with_args:nn #1 #2 { }
198 <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 14.)

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

```
\hook_new_pair_with_args:nnn
199 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_new_pair_with_args:nnn}
200 <latexrelease> {Hooks~with~args}
201 \cs_new_protected:Npn \hook_new_pair:nn #1#2
202 { \__hook_normalize_hook_args:Nnn \__hook_new_pair:nnn {#1} {#2} { 0 } }
203 \cs_new_protected:Npn \hook_new_pair_with_args:nnn #1#2#3
204 { \__hook_normalize_hook_args:Nnn \__hook_new_pair:nnn {#1} {#2} {#3} }
205 \cs_new_protected:Npn \__hook_new_pair:nnn #1 #2 #3
206 {
207   \__hook_if_declared:nTF {#1}
208   { \msg_error:nnn { hooks } { exists } {#1} }
209   {
210     \__hook_if_declared:nTF {#2}
211     { \msg_error:nnn { hooks } { exists } {#2} }
212     {
213       \__hook_new:nn {#1} {#3}
214       \__hook_new_reversed:nn {#2} {#3}
215     }
216   }
217 }
```

```

217 }
218 \<latexrelease>\EndIncludeInRelease
219 \<latexrelease>\IncludeInRelease{2020/10/01}{\hook_new_pair_with_args:nnn}
220 \<latexrelease>{Hooks~with~args}
221 \<latexrelease>\cs_gset_protected:Npn \hook_new_pair:nn #1#2
222 \<latexrelease>{
223 \<latexrelease>\hook_new:n {#1}
224 \<latexrelease>\hook_new_reversed:n {#2}
225 \<latexrelease>}
226 \<latexrelease>\cs_gset_protected:Npn \hook_new_pair_with_args:nnn #1#2#3
227 \<latexrelease>{ }
228 \<latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_new_pair:nn` and `\hook_new_pair_with_args:nnn`. These functions are documented on page 14.)

`_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!

```

229 \<latexrelease>\IncludeInRelease{2023/06/01}{\_hook_include_legacy_code_chunk:n}
230 \<latexrelease>{Hooks~with~args}
231 \cs_new_protected:Npn \_hook_include_legacy_code_chunk:n #1
232 {

```

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

```

233 \tl_if_exist:cT { @#1hook }
234 {

```

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

```

235 \tl_if_empty:cF { @#1hook }
236 {

```

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

```

237 \_hook_replacing_args_false:
238 \use:e
239 {
240 \_hook_hook_gput_code_do:nnn {#1} { legacy }
241 { \exp_not:v { @#1hook } }
242 }
243 \_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.

```

244 \_hook_tl_gclear:c { @#1hook }
245 }
246 }
247 }
248 \<latexrelease>\EndIncludeInRelease
249 \<latexrelease>\IncludeInRelease{2020/10/01}{\_hook_include_legacy_code_chunk:n}

```

```

250 <latexrelease> {Hooks~with~args}
251 <latexrelease> \cs_gset_protected:Npn \__hook_include_legacy_code_chunk:n #1
252 <latexrelease> {
253 <latexrelease> \tl_if_exist:cT { @#1hook }
254 <latexrelease> {
255 <latexrelease> \tl_if_empty:cF { @#1hook }
256 <latexrelease> {
257 <latexrelease> \exp_args:Nnnv \__hook_hook_gput_code_do:nnn
258 <latexrelease> {#1} { legacy } { @#1hook }
259 <latexrelease> \__hook_tl_gclear:c { @#1hook }
260 <latexrelease> }
261 <latexrelease> }
262 <latexrelease> }
263 <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_if_disabled_p:n` 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.

```

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

266 \cs_new_protected:Npn \hook_disable_generic:n #1
267 { \__hook_normalize_hook_args:Nn \__hook_disable:n {#1} }
268 \cs_new_protected:Npn \__hook_disable:n #1
269 {
270 \tl_gclear_new:c { g__hook_#1_declared_tl }
271 \cs_undefine:c { __hook-#1 }
272 }
273 \prg_new_conditional:Npnn \__hook_if_disabled:n #1 { p, T, F, TF }
274 {
275 \bool_lazy_and:nnTF
276 { \tl_if_exist_p:c { g__hook_#1_declared_tl } }
277 { ! \cs_if_exist_p:c { __hook-#1 } }
278 { \prg_return_true: }
279 { \prg_return_false: }
280 }
281 <latexrelease> \EndIncludeInRelease

282 <latexrelease> \IncludeInRelease{2020/10/01}{\hook_disable_generic:n}
283 <latexrelease> {Disable~hooks}
284 <latexrelease>
285 <latexrelease> \cs_new_protected:Npn \hook_disable_generic:n #1 {}
286 <latexrelease>
287 <latexrelease> \EndIncludeInRelease

```

(End of definition for `\hook_disable_generic:n`, `__hook_disable:n`, and `__hook_if_disabled:nTF`. This function is documented on page 14.)

`\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.

```

288 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_activate_generic:n}
289 <latexrelease>          {Providing-hooks}

290 \cs_new_protected:Npn \hook_activate_generic:n #1
291   { \__hook_normalize_hook_args:Nn \__hook_activate_generic:nn {#1} { } }

292 \cs_new_protected:Npn \__hook_activate_generic:nn #1 #2
293   {

```

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

```

294     \__hook_if_disabled:nTF {#1}
295     { \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.

```

296     {
297       \__hook_if_declared:nF {#1}
298       {
299         \tl_new:c { g__hook_#1_declared_tl }
300         \__hook_make_usable:nn {#1} { 0 }
301         \tl_gset:cx { g__hook_#1_reversed_tl }
302         { \__hook_if_generic_reversed:nT {#1} { - } }

```

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

```

303         \__hook_update_hook_code:n {#1}
304       }
305     }
306   }

307 <latexrelease>\EndIncludeInRelease

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

326 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_activate_generic:n}
327 <latexrelease>          {Providing-hooks}
328 <latexrelease>\cs_gset_protected:Npn \hook_activate_generic:n #1 { }
329 <latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_activate_generic:n` and `__hook_activate_generic:n`. This function is documented on page 14.)

4.5 Parsing a label

`__hook_parse_label_default:n` This macro checks if a label was given (not `\c_novalue_tl`), and if so, tries to parse the label looking for a leading `.` to replace by `__hook_currname_or_default:.`

```

330 \cs_new:Npn \__hook_parse_label_default:n #1
331 {
332   \tl_if_novalue:nTF {#1}
333     { \__hook_currname_or_default: }
334     { \tl_trim_spaces_apply:nN {#1} \__hook_parse_dot_label:n }
335 }
```

(End of definition for `__hook_parse_label_default:n`.)

`__hook_parse_dot_label:n` 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.

```

336 \cs_new:Npn \__hook_parse_dot_label:n #1
337 {
338   \tl_if_empty:nTF {#1}
339   {
340     \msg_expandable_error:nn { hooks } { empty-label }
341     \__hook_currname_or_default:
342   }
343   {
344     \str_if_eq:nnTF {#1} { . }
345     { \__hook_currname_or_default: }
346     { \__hook_parse_dot_label:w #1 ./ \s__hook_mark }
347   }
348 }
349 \cs_new:Npn \__hook_parse_dot_label:w #1 ./ #2 \s__hook_mark
350 {
351   \tl_if_empty:nTF {#1}
352   { \__hook_parse_dot_label_aux:w #2 \s__hook_mark }
353   {
354     \tl_if_empty:nTF {#2}
355     { \__hook_make_name:n {#1} }
356     { \__hook_parse_dot_label_cleanup:w #1 ./ #2 \s__hook_mark }
357   }
358 }
359 \cs_new:Npn \__hook_parse_dot_label_cleanup:w #1 ./ \s__hook_mark {#1}
360 \cs_new:Npn \__hook_parse_dot_label_aux:w #1 ./ \s__hook_mark
361 { \__hook_currname_or_default: / \__hook_make_name:n {#1} }
```

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

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

```

362 \cs_new:Npn \__hook_currname_or_default:
363 {
364   \tl_if_empty:NTF \g__hook_hook_curr_name_tl
```

```

365 {
366   \tl_if_empty:NTF \@currname
367   {
368     \msg_expandable_error:nnn { latex2e } { should-not-happen }
369     { Empty~default~label. }
370     \__hook_make_name:n { label-missing }
371   }
372   { \@currname }
373 }
374 { \g__hook_hook_curr_name_tl }
375 }

```

(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.

```

376 \cs_new:Npn \__hook_make_name:n #1
377 {
378   \exp_after:wN \exp_after:wN \exp_after:wN \__hook_make_name:w
379   \exp_after:wN \token_to_str:N \cs:w __hook~ #1 \cs_end:
380 }
381 \exp_last_unbraced:NNNNo
382 \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`!

```

383 \cs_new_protected:Npn \__hook_normalize_hook_args_aux:Nn #1 #2
384 {
385   \group_begin:
386   \use:e
387   {
388     \group_end:
389     \exp_not:N #1 #2
390   }
391 }
392 \cs_new_protected:Npn \__hook_normalize_hook_args:Nn #1 #2
393 {
394   \__hook_normalize_hook_args_aux:Nn #1
395   { { \__hook_parse_label_default:n {#2} } }
396 }
397 \cs_new_protected:Npn \__hook_normalize_hook_args:Nnn #1 #2 #3
398 {
399   \__hook_normalize_hook_args_aux:Nn #1
400   {
401     { \__hook_parse_label_default:n {#2} }
402     { \__hook_parse_label_default:n {#3} }

```

```

403     }
404   }
405   \cs_new_protected:Npn \__hook_normalize_hook_rule_args:Nnnnn #1 #2 #3 #4 #5
406   {
407     \__hook_normalize_hook_args_aux:Nn #1
408     {
409       { \__hook_parse_label_default:n {#2} }
410       { \__hook_parse_label_default:n {#3} }
411       { \tl_trim_spaces:n {#4} }
412       { \__hook_parse_label_default:n {#5} }
413     }
414   }

```

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

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

To pull that one off, we keep a stack that contains the default label for each level of input. The bottom of the stack contains the default label for the `top-level` (this stack should never go empty). If we're building the format, set the default label to be `top-level`:

```

415 \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:

```

416 <latexrelease>\seq_if_empty:NT \g__hook_name_stack_seq
417 <latexrelease> { \seq_gput_right:Nn \g__hook_name_stack_seq { top-level } }

```

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

```

418 <latexrelease>\cs_set_protected:Npn \__hook_tmp:w #1 #2 #3
419 <latexrelease> {
420 <latexrelease>   \quark_if_recursion_tail_stop:n {#1}
421 <latexrelease>   \seq_gput_right:Nn \g__hook_name_stack_seq {#1}
422 <latexrelease>   \__hook_tmp:w
423 <latexrelease> }
424 <latexrelease>\exp_after:wN \__hook_tmp:w \@currnamestack
425 <latexrelease> \q_recursion_tail \q_recursion_tail
426 <latexrelease> \q_recursion_tail \q_recursion_stop

```

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

```

427 <latexrelease>\tl_gset:Nx \g__hook_hook_curr_name_tl { \@currname }
428 <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):

```

429 \cs_new_protected:Npn \__hook_curr_name_push:n #1
430 { \exp_args:Nx \__hook_curr_name_push_aux:n { \__hook_make_name:n {#1} } }
431 \cs_new_protected:Npn \__hook_curr_name_push_aux:n #1
432 {
433   \tl_if_blank:nTF {#1}
434   { \msg_error:nn { hooks } { no-default-label } }

```

```

435     {
436       \str_if_eq:nnTF {#1} { top-level }
437       {
438         \msg_error:nnnnn { hooks } { set-top-level }
439         { to } { PushDefaultHookLabel } {#1}
440       }
441       {
442         \seq_gpush:NV \g__hook_name_stack_seq \g__hook_hook_curr_name_tl
443         \tl_gset:Nn \g__hook_hook_curr_name_tl {#1}
444       }
445     }
446   }

```

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:

```

447 \cs_new_protected:Npn \__hook_curr_name_pop:
448 {
449   \seq_gpop:NNTF \g__hook_name_stack_seq \l__hook_return_tl
450   { \tl_gset_eq:NN \g__hook_hook_curr_name_tl \l__hook_return_tl }
451   { \msg_error:nn { hooks } { extra-pop-label } }
452 }

```

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):

```

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

```

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.

```

465 \cs_new_protected:Npn \__hook_set_default_hook_label:n #1
466 {
467   \seq_if_empty:NNTF \g__hook_name_stack_seq
468   {
469     \msg_error:nnnnn { hooks } { set-top-level }
470     { for } { SetDefaultHookLabel } {#1}
471   }
472   { \exp_args:Nx \__hook_set_default_label:n { \__hook_make_name:n {#1} } }
473 }
474 \cs_new_protected:Npn \__hook_set_default_label:n #1
475 {
476   \str_if_eq:nnTF {#1} { top-level }

```

```

477     {
478       \msg_error:nnnnn { hooks } { set-top-level }
479       { to } { SetDefaultHookLabel } {#1}
480     }
481     { \tl_gset:Nn \g__hook_hook_curr_name_tl {#1} }
482   }

```

(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
483 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_gput_code:nnn}
484 <latexrelease>{Hooks~with~args}
485 \cs_new_protected:Npn \hook_gput_code:nnn #1 #2 #3
486 {
487   \__hook_replacing_args_false:
488   \__hook_normalize_hook_args:Nnn \__hook_gput_code:nnn {#1} {#2} {#3}
489   \__hook_replacing_args_reset:
490 }
491 \cs_new_protected:Npn \hook_gput_code_with_args:nnn #1 #2 #3
492 {
493   \__hook_replacing_args_true:
494   \__hook_normalize_hook_args:Nnn \__hook_gput_code:nnn {#1} {#2} {#3}
495   \__hook_replacing_args_reset:
496 }

```

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:`.

```

497 \cs_new_protected:Npn \__hook_gput_code:nnn #1 #2 #3
498 {
499   \__hook_chk_args_allowed:nn {#1} { AddToHook }

```

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

```

500   \__hook_if_execute_immediately:nTF {#1}
501   {

```

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

```

502     \__hook_if_replacing_args:TF
503     {
504       \msg_error:nnnn { hooks } { one-time-args }
505       {#1} { AddToHook }
506     }
507     { }
508     \use:n
509   }
510   { \__hook_gput_code_store:nnn {#1} {#2} }
511     {#3}
512 }
513 \cs_new_protected:Npn \__hook_gput_code_store:nnn #1 #2 #3
514 {

```

Then check if the hook is usable.

```
515 \__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.

```
516 {
517 \__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.

```
518 \__hook_update_hook_code:n {#1}
519 }
```

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.

```
520 {
521 \__hook_if_disabled:nTF {#1}
522 { \msg_error:nnn { hooks } { hook-disabled } {#1} }
523 { \__hook_try_declaring_generic_hook:nnn {#1} {#2} {#3} }
524 }
525 }
```

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

```
526 \cs_new_protected:Npn \__hook_hook_gput_code_do:nnn #1 #2 #3
527 {
```

However, first some debugging info if debugging is enabled:

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

Then try to get the code chunk labeled `#2` from the hook. If there's code already there, then append `#3` to that, otherwise just put `#3`. If the current label is `top-level`, the code is added to a dedicated token list `__hook_toplevel_<hook>` that goes at the end of the hook (or at the beginning, for a reversed hook), just before `__hook_next_<hook>`.

```
532 \str_if_eq:nnTF {#2} { top-level }
533 {
534 \str_if_eq:eeTF { top-level } { \__hook_currname_or_default: }
535 {
```

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

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

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

```
537 \__hook_cs_gput_right:nnn { _toplevel } {#1} {#3}
538 }
539 { \msg_error:nnn { hooks } { misused-top-level } {#1} }
540 }
541 {
```

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.

```
542 \exp_args:Nx \__hook_prop_gput_labeled_cleanup:nnn
```

```

543         {
544             \__hook_if_replacing_args:TF
545             { \exp_not:n }
546             { \__hook_double_hashes:n }
547             {#3}
548         }
549         {#1} {#2}
550     }
551 }

Adds code to a hook's code pool.

552 \cs_new_protected:Npn \__hook_prop_gput_labeled_cleanup:nnn #1 #2 #3
553 {
554     \tl_set:Nn \l__hook_return_tl {#1}
555     \__hook_if_replacing_args:TF
556     {
557         \__hook_if_usable:nT {#2}
558         {
559             \__hook_set_normalise_fn:nn {#2}
560             { Invalid-code-added~\msg_line_context: }
561             \__hook_normalise_fn:nn {#3} {#1}
562             \prop_get:NnN \l__hook_work_prop {#3} \l__hook_return_tl
563         }
564     }
565     { }
566     \exp_args:NcV \__hook_prop_gput_labeled_do:Nnn
567     { g__hook_#2_code_prop } \l__hook_return_tl {#3}
568 }
569 \cs_new_protected:Npn \__hook_prop_gput_labeled_do:Nnn #1 #2 #3
570 {
571     \prop_get:NnNTF #1 {#3} \l__hook_return_tl
572     { \prop_gput:Nno #1 {#3} { \l__hook_return_tl #2 } }
573     { \prop_gput:Nnn #1 {#3} {#2} }
574 }

575 <latexrelease>\EndIncludeInRelease

576 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_gput_code:nnn}
577 <latexrelease>          {Providing-hooks}
578 <latexrelease>\cs_gset_protected:Npn \hook_gput_code:nnn #1 #2
579 <latexrelease> { \__hook_normalize_hook_args:Nnn \__hook_gput_code:nnn {#1} {#2} }
580 <latexrelease>\cs_gset_protected:Npn \__hook_gput_code:nnn #1 #2 #3
581 <latexrelease> {
582 <latexrelease>     \__hook_if_execute_immediately:nTF {#1}
583 <latexrelease>     {#3}
584 <latexrelease>     {
585 <latexrelease>         \__hook_if_usable:nTF {#1}
586 <latexrelease>         {
587 <latexrelease>             \__hook_hook_gput_code_do:nnn {#1} {#2} {#3}
588 <latexrelease>             \__hook_update_hook_code:n {#1}
589 <latexrelease>         }
590 <latexrelease>         {
591 <latexrelease>             \__hook_if_disabled:nTF {#1}
592 <latexrelease>             { \msg_error:nnn { hooks } { hook-disabled } {#1} }
593 <latexrelease>             { \__hook_try_declaring_generic_hook:nnn {#1} {#2} {#3} }
594 <latexrelease>         }

```



```

595 <latexrelease>      }
596 <latexrelease>    }
597 <latexrelease> \cs_gset_protected:Npn \__hook_hook_gput_code_do:nnn #1 #2 #3
598 <latexrelease> {
599 <latexrelease>   \__hook_debug:n{\iow_term:x{****~ Add~ to~
600 <latexrelease>               \__hook_if_usable:nF {#1} { undeclared~ }
601 <latexrelease>               hook~ #1~ (#2)
602 <latexrelease>               \on@line\space <~ \tl_to_str:n{#3}} }
603 <latexrelease>   \str_if_eq:nnTF {#2} { top-level }
604 <latexrelease>   {
605 <latexrelease>     \str_if_eq:eeTF { top-level } { \__hook_currname_or_default: }
606 <latexrelease>     {
607 <latexrelease>       \__hook_init_structure:n {#1}
608 <latexrelease>       \__hook_tl_gput_right:cn { \__hook_toplevel~#1 } {#3}
609 <latexrelease>     }
610 <latexrelease>     { \msg_error:nnn { hooks } { misused-top-level } {#1} }
611 <latexrelease>   }
612 <latexrelease>   {
613 <latexrelease>     \prop_get:cnNTF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
614 <latexrelease>     {
615 <latexrelease>       \prop_gput:cno { g__hook_#1_code_prop } {#2}
616 <latexrelease>       { \l__hook_return_tl #3 }
617 <latexrelease>     }
618 <latexrelease>     { \prop_gput:cnn { g__hook_#1_code_prop } {#2} {#3} }
619 <latexrelease>   }
620 <latexrelease> }
621 <latexrelease> \cs_gset_protected:Npn \hook_gput_code_with_args:nnn #1#2#3 { }
622 <latexrelease> \EndIncludeInRelease

```

(End of definition for \hook_gput_code:nnn and others. These functions are documented on page 15.)

`__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.

```

623 <latexrelease> \IncludeInRelease{2023/06/01}{\__hook_chk_args_allowed:nn}
624 <latexrelease>   {Hooks~with~args}
625 <latexrelease> \cs_new_protected:Npn \__hook_chk_args_allowed:nn #1 #2
626 <latexrelease> {
627 <latexrelease>   \__hook_if_replacing_args:TF
628 <latexrelease>   {
629 <latexrelease>     \__hook_if_declared:nT {#1}
630 <latexrelease>     { \tl_if_empty:cT { c__hook_#1_parameter_tl } { \use_ii:nn } }
631 <latexrelease>     \use_none:n
632 <latexrelease>     {
633 <latexrelease>       \msg_error:nnnn { hooks } { without-args } {#1} {#2}
634 <latexrelease>       \__hook_replacing_args_false:
635 <latexrelease>     }
636 <latexrelease>   }
637 <latexrelease> { }
638 <latexrelease> }
639 <latexrelease> \EndIncludeInRelease
640 <latexrelease> \IncludeInRelease{2020/10/01}{\__hook_chk_args_allowed:nn}

```

```

641 <latexrelease> {Hooks~with~args}
642 <latexrelease>\cs_undefine:N \__hook_chk_args_allowed:nn
643 <latexrelease>\EndIncludeInRelease

```

(End of definition for __hook_chk_args_allowed:nn.)

__hook_gput_undeclared_hook:nnn

Often it may happen that a package *A* defines a hook `foo`, but package *B*, that adds code to that hook, is loaded before *A*. In such case we need to add code to the hook before its declared. An implicitly declared hook doesn't have arguments (in principle), so use `\c_false_bool` here.

```

644 \cs_new_protected:Npn \__hook_gput_undeclared_hook:nnn #1 #2 #3
645 {
646   \__hook_init_structure:n {#1}
647   \__hook_hook_gput_code_do:nnn {#1} {#2} {#3}
648 }

```

(End of definition for __hook_gput_undeclared_hook:nnn.)

__hook_try_declaring_generic_hook:nnn

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.

```

649 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_try_declaring_generic_hook:nnn}
650 <latexrelease> {Hooks~with~args}
651 \cs_new_protected:Npn \__hook_try_declaring_generic_hook:nnn #1
652 {
653   \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
654   \__hook_gput_code:nnn
655   \__hook_gput_undeclared_hook:nnn
656   {#1}
657 }
658 \cs_new_protected:Npn \__hook_try_declaring_generic_next_hook:nn #1
659 {
660   \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
661   \__hook_gput_next_code:nn
662   \__hook_gput_next_do:nn
663   {#1}
664 }
665 <latexrelease>\EndIncludeInRelease
666 <latexrelease>\IncludeInRelease{2021/11/15}{\__hook_try_declaring_generic_hook:nnn}
667 <latexrelease> {Standardise~generic~hook~names}
668 <latexrelease>\cs_gset_protected:Npn \__hook_try_declaring_generic_hook:nnn #1
669 <latexrelease> {
670 <latexrelease>   \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
671 <latexrelease>   \hook_gput_code:nnn
672 <latexrelease>   \__hook_gput_undeclared_hook:nnn
673 <latexrelease>   {#1}

```

```

674 <latexrelease> }
675 <latexrelease> \cs_gset_protected:Npn \__hook_try_declaring_generic_next_hook:nn #1
676 <latexrelease> {
677 <latexrelease>     \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
678 <latexrelease>     \hook_gput_next_code:nn
679 <latexrelease>     \__hook_gput_next_do:nn
680 <latexrelease>     {#1}
681 <latexrelease> }
682 <latexrelease> \EndIncludeInRelease
683 <latexrelease> \IncludeInRelease{2020/10/01}{\__hook_try_declaring_generic_hook:nnn}
684 <latexrelease>     {Standardise-generic-hook-names}
685 <latexrelease> \cs_new_protected:Npn \__hook_try_declaring_generic_hook:nnn #1
686 <latexrelease> {
687 <latexrelease>     \__hook_try_declaring_generic_hook:nNNnn {#1}
688 <latexrelease>     \hook_gput_code:nnn \__hook_gput_undeclared_hook:nnn
689 <latexrelease> }
690 <latexrelease> \cs_new_protected:Npn \__hook_try_declaring_generic_next_hook:nn #1
691 <latexrelease> {
692 <latexrelease>     \__hook_try_declaring_generic_hook:nNNnn {#1}
693 <latexrelease>     \hook_gput_next_code:nn \__hook_gput_next_do:nn
694 <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 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.

```

695 <latexrelease> \cs_new_protected:Npn \__hook_try_declaring_generic_hook:nNNnn #1
696 <latexrelease> {
697 <latexrelease>     \__hook_if_file_hook:wTF #1 / \s_hook_mark
698 <latexrelease>     {
699 <latexrelease>         \exp_args:Ne \__hook_try_declaring_generic_hook_split:nNNnn
700 <latexrelease>         { \exp_args:Ne \__hook_file_hook_normalize:n {#1} }
701 <latexrelease>     }
702 <latexrelease>     { \__hook_try_declaring_generic_hook_split:nNNnn {#1} }
703 <latexrelease> }
704 <latexrelease> \cs_new_protected:Npn \__hook_try_declaring_generic_hook_split:nNNnn #1 #2 #3
705 <latexrelease> {
706 <latexrelease>     \__hook_try_declaring_generic_hook:wnTF #1 / / / \scan_stop: {#1}
707 <latexrelease>     { #2 }
708 <latexrelease>     { #3 } {#1}
709 <latexrelease> }
710 <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

```

711 <latexrelease> \IncludeInRelease{2023/06/01}{\__hook_try_declaring_generic_hook:wn}
712 <latexrelease>     {Hooks-with-args}
713 \prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn

```

```

714     #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
715 {
716     \__hook_if_generic:nTF {#5}
717     {
718         \__hook_if_usable:nF {#5}
719         {

```

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.

```

720         \str_if_eq:nnT {#1} { cmd }
721         {
722             \__hook_try_put_cmd_hook:n {#5}
723             \__hook_make_usable:nn {#5} { 9 }
724             \use_none:nnn
725         }

```

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.

```

726         \__hook_make_usable:nn {#5} { 0 }
727     }
728     \__hook_if_generic_reversed:nT {#5}
729     { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
730     \prg_return_true:
731 }
732 {

```

Generic hooks are all named $\langle type \rangle / \langle name \rangle / \langle place \rangle$, where $\langle type \rangle$ and $\langle place \rangle$ are predefined (`\c__hook_generic_⟨type⟩/.⟨place⟩_tl`), and $\langle name \rangle$ is the variable component. Older releases had some hooks with the $\langle name \rangle$ 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:.`

```

733     \__hook_if_deprecated_generic:nTF {#5}
734     {
735         \__hook_deprecated_generic_warn:n {#5}
736         \exp_after:wN \__hook_declare_deprecated_generic:NNn
737         \exp:w % \exp_end:
738     }
739     { \prg_return_false: }
740 }
741 }

```

`__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

```

```

742 \cs_new_protected:Npn \__hook_deprecated_generic_warn:n #1
743 { \__hook_deprecated_generic_warn:w #1 \s__hook_mark }

```

```

744 \cs_new_protected:Npn \__hook_deprecated_generic_warn:w
745   #1 / #2 / #3 \s__hook_mark
746   {
747     \if_cs_exist:w __hook~#1/#2/#3 \cs_end: \else:
748       \msg_warning:nnnnn { hooks } { generic-deprecated } {#1} {#2} {#3}
749     \fi:
750     \cs_gset_eq:cN { __hook~#1/#2/#3 } \scan_stop:
751   }

```

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
752 \cs_new_protected:Npn \__hook_do_deprecated_generic:Nn #1 #2
753   { \__hook_do_deprecated_generic:Nw #1 #2 \s__hook_mark }
754 \cs_new_protected:Npn \__hook_do_deprecated_generic:Nw #1
755   #2 / #3 / #4 \s__hook_mark
756   { #1 { #2 / #4 / #3 } }
757 \cs_new_protected:Npn \__hook_declare_deprecated_generic:NNn #1 #2 #3
758   { \__hook_declare_deprecated_generic:NNw #1 #2 #3 \s__hook_mark }
759 \cs_new_protected:Npn \__hook_declare_deprecated_generic:NNw #1 #2
760   #3 / #4 / #5 \s__hook_mark
761   {
762     \__hook_try_declaring_generic_hook:wnTF #3 / #5 / #4 / \scan_stop:
763     { #3 / #5 / #4 }
764     #1 #2 { #3 / #5 / #4 }
765   }
766 \<latexrelease>\EndIncludeInRelease

767 \<latexrelease>\IncludeInRelease{2021/11/15}{\__hook_try_declaring_generic_hook:wn}
768 \<latexrelease>{Standardise-generic-hook-names}
769 \<latexrelease>\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
770 \<latexrelease>   #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
771 \<latexrelease>   {
772 \<latexrelease>     \__hook_if_generic:nTF {#5}
773 \<latexrelease>     {
774 \<latexrelease>       \__hook_if_usable:nF {#5}
775 \<latexrelease>       {
776 \<latexrelease>         \str_if_eq:nnT {#1} { cmd }
777 \<latexrelease>         { \__hook_try_put_cmd_hook:n {#5} }
778 \<latexrelease>         \__hook_make_usable:n {#5}
779 \<latexrelease>       }
780 \<latexrelease>       \__hook_if_generic_reversed:nT {#5}
781 \<latexrelease>       { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
782 \<latexrelease>       \prg_return_true:
783 \<latexrelease>     }
784 \<latexrelease>     {
785 \<latexrelease>       \__hook_if_deprecated_generic:nTF {#5}
786 \<latexrelease>       {
787 \<latexrelease>         \__hook_deprecated_generic_warn:n {#5}
788 \<latexrelease>         \exp_after:wN \__hook_declare_deprecated_generic:NNn
789 \<latexrelease>         \exp:w % \exp_end:
790 \<latexrelease>       }
791 \<latexrelease>       { \prg_return_false: }
792 \<latexrelease>     }
793 \<latexrelease>   }

```

```

794 <latexrelease>\EndIncludeInRelease

795 <latexrelease>\IncludeInRelease{2021/06/01}{\__hook_try_declaring_generic_hook:wn}
796 <latexrelease>          {Support~cmd~hooks}
797 <latexrelease>\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
798 <latexrelease>      #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
799 <latexrelease> {
800 <latexrelease>     \tl_if_empty:nTF {#2}
801 <latexrelease>     { \prg_return_false: }
802 <latexrelease>     {
803 <latexrelease>         \prop_if_in:NnTF \c__hook_generics_prop {#1}
804 <latexrelease>         {
805 <latexrelease>             \__hook_if_usable:nF {#5}
806 <latexrelease>             {
807 <latexrelease>                 \str_if_eq:nnT {#1} { cmd }
808 <latexrelease>                 { \__hook_try_put_cmd_hook:n {#5} }
809 <latexrelease>                 \__hook_make_usable:n {#5}
810 <latexrelease>             }
811 <latexrelease>         \prop_if_in:NnTF \c__hook_generics_reversed_ii_prop {#2}
812 <latexrelease>         { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
813 <latexrelease>         {
814 <latexrelease>             \prop_if_in:NnT \c__hook_generics_reversed_iii_prop {#3}
815 <latexrelease>             { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
816 <latexrelease>         }
817 <latexrelease>         \prg_return_true:
818 <latexrelease>     }
819 <latexrelease>     { \prg_return_false: }
820 <latexrelease> }
821 <latexrelease> }
822 <latexrelease>\EndIncludeInRelease

823 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_try_declaring_generic_hook:wn}
824 <latexrelease>          {Support~cmd~hooks}
825 <latexrelease>\prg_new_protected_conditional:Npnn \__hook_try_declaring_generic_hook:wn
826 <latexrelease>      #1 / #2 / #3 / #4 \scan_stop: #5 { TF }
827 <latexrelease> {
828 <latexrelease>     \tl_if_empty:nTF {#2}
829 <latexrelease>     { \prg_return_false: }
830 <latexrelease>     {
831 <latexrelease>         \prop_if_in:NnTF \c__hook_generics_prop {#1}
832 <latexrelease>         {
833 <latexrelease>             \__hook_if_declared:nF {#5} { \hook_new:n {#5} }
834 <latexrelease>             \prop_if_in:NnTF \c__hook_generics_reversed_ii_prop {#2}
835 <latexrelease>             { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
836 <latexrelease>             {
837 <latexrelease>                 \prop_if_in:NnT \c__hook_generics_reversed_iii_prop {#3}
838 <latexrelease>                 { \tl_gset:cn { g__hook_#5_reversed_tl } { - } }
839 <latexrelease>             }
840 <latexrelease>             \prg_return_true:
841 <latexrelease>         }
842 <latexrelease>         { \prg_return_false: }
843 <latexrelease>     }
844 <latexrelease> }
845 <latexrelease>\EndIncludeInRelease

```

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

`__hook_if_file_hook_p:w` `__hook_if_file_hook:wTF` 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`.

```

846 <latexrelease> \IncludeInRelease{2021/11/15}{\__hook_if_file_hook:w}
847 <latexrelease> {Standardise-generic-hook-names}
848 <latexrelease> \EndIncludeInRelease
849 <latexrelease> \IncludeInRelease{2020/10/01}{\__hook_if_file_hook:w}
850 <latexrelease> {Standardise-generic-hook-names}
851 <latexrelease> \prg_new_conditional:Npnn \__hook_if_file_hook:w
852 <latexrelease> #1 / #2 / #3 \s__hook_mark { TF }
853 <latexrelease> {
854 <latexrelease> \str_if_eq:nnTF {#1} { file }
855 <latexrelease> {
856 <latexrelease> \bool_lazy_or:nnTF
857 <latexrelease> { \tl_if_empty_p:n {#3} }
858 <latexrelease> { \str_if_eq_p:nn {#3} { / } }
859 <latexrelease> { \prg_return_false: }
860 <latexrelease> {
861 <latexrelease> \prop_if_in:NnTF \c__hook_generics_file_prop {#2}
862 <latexrelease> { \prg_return_true: }
863 <latexrelease> { \prg_return_false: }
864 <latexrelease> }
865 <latexrelease> }
866 <latexrelease> { \prg_return_false: }
867 <latexrelease> }
868 <latexrelease> \EndIncludeInRelease

```

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

```

\__hook_file_hook_normalize:n
\__hook_strip_double_slash:n
\__hook_strip_double_slash:w
869 <latexrelease> \IncludeInRelease{2021/11/15}{\__hook_file_hook_normalize:n}
870 <latexrelease> {Standardise-generic-hook-names}
871 <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}`.

```

872 <latexrelease> \IncludeInRelease{2020/10/01}{\__hook_file_hook_normalize:n}
873 <latexrelease> {Standardise-generic-hook-names}
874 <latexrelease> \cs_new:Npn \__hook_file_hook_normalize:n #1
875 <latexrelease> { \__hook_strip_double_slash:n {#1} }
876 <latexrelease> \cs_new:Npn \__hook_strip_double_slash:n #1
877 <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.

```

878 <latexrelease> \cs_new:Npn \__hook_strip_double_slash:w #1/#2/#3 // #4 \s__hook_mark
879 <latexrelease> {
880 <latexrelease> \tl_if_empty:nTF {#4}
881 <latexrelease> { #1/#2/#3 }

```

```

882 <latexrelease> { \__hook_strip_double_slash:w #1/#2/#3 / #4 \s__hook_mark }
883 <latexrelease> }
884 <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

```

885 <latexrelease>\IncludeInRelease{2021/11/15}{\c__hook_generics_prop}
886 <latexrelease> {Standardise-generic-hook-names}
887 \clist_map_inline:nn { cmd , env , file , package , class , include }
888 {
889   \tl_const:cn { c__hook_generic_#1/.before_tl } { + }
890   \tl_const:cn { c__hook_generic_#1/.after_tl } { - }
891 }
892 \tl_const:cn { c__hook_generic_env/.begin_tl } { + }
893 \tl_const:cn { c__hook_generic_env/.end_tl } { + }
894 \tl_const:cn { c__hook_generic_include/.end_tl } { - }
895 \tl_const:cn { c__hook_generic_include/.excluded_tl } { + }

```

Deprecated generic hooks:

```

896 \clist_map_inline:nn { file , package , class , include }
897 {
898   \tl_const:cn { c__hook_deprecated_#1/.before_tl } { }
899   \tl_const:cn { c__hook_deprecated_#1/.after_tl } { }
900 }
901 \tl_const:cn { c__hook_deprecated_include/.end_tl } { }
902 <latexrelease>\EndIncludeInRelease
903 <latexrelease>\IncludeInRelease{2020/10/01}{\c__hook_generics_prop}
904 <latexrelease> {Standardise-generic-hook-names}
905 <latexrelease>\prop_const_from_keyval:Nn \c__hook_generics_prop
906 <latexrelease> {cmd=,env=,file=,package=,class=,include=}
907 <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):

```

908 <latexrelease>\IncludeInRelease{2021/11/15}{\c__hook_generics_reversed_ii_prop}
909 <latexrelease> {Standardise-generic-hook-names}
910 <latexrelease>\EndIncludeInRelease
911 <latexrelease>\IncludeInRelease{2020/10/01}{\c__hook_generics_reversed_ii_prop}
912 <latexrelease> {Standardise-generic-hook-names}
913 <latexrelease>\prop_const_from_keyval:Nn \c__hook_generics_reversed_ii_prop {after=,end=}
914 <latexrelease>\prop_const_from_keyval:Nn \c__hook_generics_reversed_iii_prop {after=}
915 <latexrelease>\prop_const_from_keyval:Nn \c__hook_generics_file_prop {before=,after=}
916 <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.

```
917 <latexrelease>\IncludeInRelease{2023/06/01}{\c__hook_parameter_cmd/.before_tl}
918 <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.

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

921 <latexrelease>\EndIncludeInRelease
922 <latexrelease>\IncludeInRelease{2020/10/01}{\c__hook_parameter_cmd/.before_tl}
923 <latexrelease>{Hooks~with~args}
924 <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.

```
925 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_gremove_code:nn}
926 <latexrelease>{Hooks~with~args}
927 \cs_new_protected:Npn \hook_gremove_code:nn #1 #2
928 { \__hook_normalize_hook_args:Nnn \__hook_gremove_code:nn {#1} {#2} }
929 \cs_new_protected:Npn \__hook_gremove_code:nn #1 #2
930 {
```

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).

```
931 \__hook_if_structure_exist:nTF {#1}
932 {
```

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>`.

```
933 \str_if_eq:nnTF {#2} {*}
934 {
935 \prop_gclear:c { g__hook_#1_code_prop }
936 \__hook_toplevel_gset:nn {#1} { }
937 \__hook_next_gset:nn {#1} { }
938 }
939 {
```

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

```
940 \str_if_eq:nnTF {#2} { top-level }
941 { \__hook_toplevel_gset:nn {#1} { } }
942 {
943 \prop_gpop:cnNF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
944 { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
```

```

945         }
946     }

```

Finally update the code, if the hook exists.

```

947     \_hook_if_usable:nT {#1}
948     { \_hook_update_hook_code:n {#1} }
949 }

```

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

```

950 {
951     \_hook_if_deprecated_generic:nTF {#1}
952     {
953         \_hook_deprecated_generic_warn:n {#1}
954         \_hook_do_deprecated_generic:Nn \_hook_gremove_code:nn {#1} {#2}
955     }
956     { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
957 }
958 }
959 \<latexrelease>\EndIncludeInRelease

960 \<latexrelease>\IncludeInRelease{2020/10/01}{\hook_gremove_code:nn}
961 \<latexrelease>{Hooks~with~args}
962 \<latexrelease>\cs_new_protected:Npn \_hook_gremove_code:nn #1 #2
963 \<latexrelease> {
964     \_hook_if_structure_exist:nTF {#1}
965     {
966         \str_if_eq:nnTF {#2} {*}
967         {
968             \prop_gclear:c { g__hook_#1_code_prop }
969             \_hook_tl_gclear:c { __hook_toplevel~#1 }
970             \_hook_tl_gclear:c { __hook_next~#1 }
971         }
972         {
973             \str_if_eq:nnTF {#2} { top-level }
974             { \_hook_tl_gclear:c { __hook_toplevel~#1 } }
975             {
976                 \prop_gpop:cnNF { g__hook_#1_code_prop } {#2} \l__hook_return_tl
977                 { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
978             }
979         }
980         \_hook_if_usable:nT {#1}
981         { \_hook_update_hook_code:n {#1} }
982     }
983     {
984         \_hook_if_deprecated_generic:nTF {#1}
985         {
986             \_hook_deprecated_generic_warn:n {#1}
987             \_hook_do_deprecated_generic:Nn \_hook_gremove_code:nn {#1} {#2}
988         }
989         { \msg_warning:nnnn { hooks } { cannot-remove } {#1} {#2} }
990     }
991 }
992 \<latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_gremove_code:nn` and `_hook_gremove_code:nn`. This function is documented on page 16.)

```

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

```

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

```

993 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_cs_gput_right:nnn}
994 <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.

```

995 \cs_new_protected:Npn \__hook_cs_gput_right:nnn #1 #2
996 {
997   \if:w T
998     \__hook_if_declared:nF {#2} { F }
999     \tl_if_empty:cF { c__hook_#2_parameter_tl } { F }
1000     T
1001     \exp_after:wN \__hook_cs_gput_right_fast:nnn
1002   \else:
1003     \exp_after:wN \__hook_cs_gput_right_slow:nnn
1004   \fi:
1005   {#1} {#2}
1006 }
1007 \cs_new_protected:Npn \__hook_cs_gput_right_fast:nnn #1 #2 #3
1008 { \cs_gset:cpx { __hook#1~#2 } { \exp_not:v { __hook#1~#2 } \exp_not:n {#3} } }
1009 \cs_new_protected:Npn \__hook_cs_gput_right_slow:nnn #1 #2 #3
1010 {

```

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.

```

1011 \cs_if_exist:cF { __hook#1~#2 }
1012 { \__hook_code_gset_aux:nnn {#1} {#2} { } }
1013 \__hook_code_gset_auxi:eeen
1014 {
1015   \__hook_if_declared:nTF {#2}
1016   { \tl_use:c { c__hook_#2_parameter_tl } }
1017   {
1018     \__hook_if_generic:nTF {#2}
1019     { \__hook_generic_parameter:n {#2} }
1020     { \c__hook_nine_parameters_tl }
1021   }
1022 }

```

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

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

```

1023 {
1024   \exp_args:NNo \exp_args:No \__hook_double_hashes:n
1025   {
1026     \cs:w __hook#1~#2 \exp_last_unbraced:Ne \cs_end:
1027     { \__hook_braced_cs_parameter:n { __hook#1~#2 } }
1028   }
1029 }

```

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

```

1030 {

```

```

1031     \__hook_if_replacing_args:TF
1032     { \exp_not:n }
1033     { \__hook_double_hashes:n }
1034     {#3}
1035 }

```

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

```

1036 { __hook#1~#2 }
1037 }

```

And as promised, the auxiliary that does the definition.

```

1038 \cs_new_protected:Npn \__hook_code_gset_auxi:nnnn #1 #2 #3 #4
1039 { \cs_gset:cpn {#4} #1 { #2 #3 } }
1040 \cs_generate_variant:Nn \__hook_code_gset_auxi:nnnn { een }

1041 <latexrelease>\EndIncludeInRelease
1042 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_cs_gput_right:nnn}
1043 <latexrelease>{Hooks~with~args}
1044 <latexrelease>\cs_undefine:N \__hook_cs_gput_right:nnn
1045 <latexrelease>\cs_undefine:N \__hook_cs_gput_right_fast:nnn
1046 <latexrelease>\cs_undefine:N \__hook_cs_gput_right_slow:nnn
1047 <latexrelease>\cs_undefine:N \__hook_code_gset_auxi:nnnn
1048 <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
1049 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_code_gset:nn}
1050 <latexrelease>{Hooks~with~args}
1051 \cs_new_protected:Npn \__hook_code_gset:nn
1052 { \__hook_code_gset_aux:nnn { } }
1053 \cs_new_protected:Npn \__hook_toplevel_gset:nn
1054 { \__hook_code_gset_aux:nnn { _toplevel } }
1055 \cs_new_protected:Npn \__hook_next_gset:nn
1056 { \__hook_code_gset_aux:nnn { _next } }
1057 \cs_new_protected:Npn \__hook_code_gset_aux:nnn #1 #2 #3
1058 {
1059     \cs_gset:cpn { __hook#1~#2 \exp_last_unbraced:Ne }
1060     { \__hook_parameter:n {#2} }
1061     {#3}
1062 }
1063 \cs_generate_variant:Nn \__hook_code_gset:nn { ne }

1064 <latexrelease>\EndIncludeInRelease
1065 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_code_gset:nn}
1066 <latexrelease>{Hooks~with~args}
1067 <latexrelease>\cs_undefine:N \__hook_code_gset:nn
1068 <latexrelease>\cs_undefine:N \__hook_toplevel_gset:nn
1069 <latexrelease>\cs_undefine:N \__hook_next_gset:nn
1070 <latexrelease>\cs_undefine:N \__hook_code_gset_aux:nnn
1071 <latexrelease>\EndIncludeInRelease

```

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

`_hook_normalise_cs_args:nn` This macro normalises the parameters of the macros `_hook<type>_<hook>` to take the right number of arguments after a hook is declared. At this point we know `\c_hook_<hook>_parameter_tl` exists, so use that to count the arguments and use that as *<parameter text>* for the newly (re)defined macro.

```

1072 <latexrelease>\IncludeInRelease{2023/06/01}{\_hook_normalise_cs_args:nn}
1073 <latexrelease>{Hooks~with~args}
1074 \cs_new_protected:Npn \_hook_normalise_cs_args:nn #1 #2
1075 {
1076   \cs_if_exist:cT { \_hook#1~#2 }
1077   {
1078     \_hook_code_gset_auxi:eeen
1079     { \tl_use:c { c\_hook_#2_parameter_tl } }
1080     {
1081       \exp_args:NNo \exp_args:No \_hook_double_hashes:n
1082       {
1083         \cs:w \_hook#1~#2 \exp_last_unbraced:Ne \cs_end:
1084         { \_hook_braced_cs_parameter:n { \_hook#1~#2 } }
1085       }
1086     }
1087     { }
1088     { \_hook#1~#2 }
1089   }
1090 }
1091 <latexrelease>\EndIncludeInRelease

1092 <latexrelease>\IncludeInRelease{2020/10/01}{\_hook_normalise_cs_args:nn}
1093 <latexrelease>{Hooks~with~args}
1094 <latexrelease>\cs_undefine:N \_hook_normalise_cs_args:nn
1095 <latexrelease>\EndIncludeInRelease

```

(End of definition for `_hook_normalise_cs_args:nn`.)

`_hook_normalise_code_pool:n`
`_hook_set_normalise_fn:nn` This one's a bit of a hack. It takes a hook, and iterates over its code pool (`\g_hook_<hook>_code_prop`), redefining each code label to use only valid arguments. This is used when, for example, a code is added referencing arguments #1 and #2, but the hook has only #1. In this example, every reference to #2 is changed to ##2. This is done because otherwise 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.

```

1096 <latexrelease>\IncludeInRelease{2023/06/01}{\_hook_normalise_code_pool:n}
1097 <latexrelease>{Hooks~with~args}
1098 \cs_new_protected:Npn \_hook_normalise_code_pool:n #1
1099 {

```

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

```

1100   \_hook_set_normalise_fn:nn {#1} { Offending~label:~'##1' }
1101   \prop_clear:N \l\_hook_work_prop
1102   \prop_map_function:cN { g\_hook_#1_code_prop } \_hook_normalise_fn:nn
1103   \prop_gset_eq:cN { g\_hook_#1_code_prop } \l\_hook_work_prop
1104 }

```

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

```

1105 \cs_new_protected:Npn \__hook_set_normalise_fn:nn #1 #2
1106 {

```

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}

1107 \cs_set:Npn \__hook_tmp:w ##1##2##3##4##5##6##7##8##9 { }
1108 \tl_set:Nc \l__hook_tmpb_tl
1109 { \__hook_braced_cs_parameter:n { __hook_tmp:w } }
1110 \group_begin:
1111 \__hook_tl_set:cn { c__hook_hash_tl } { \exp_not:N \c__hook_hashes_tl }
1112 \use:e
1113 {
1114 \group_end:
1115 \tl_set:Nn \exp_not:N \l__hook_tmpb_tl { \l__hook_tmpb_tl }
1116 }

```

And `\l__hook_tmpa_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.

```

1117 \exp_last_unbraced:NNf
1118 \cs_set:Npn \__hook_tmp:w { \__hook_parameter:n {#1} } { }
1119 \tl_set:Nc \l__hook_tmpa_tl { \__hook_braced_cs_parameter:n { __hook_tmp:w } }

```

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

```

1120 \cs_gset_protected:Npx \__hook_normalise_fn:nn ##1 ##2
1121 {

```

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

```

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

```

3

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

```

1122 \group_begin:
1123 \int_set:Nn \tex_escapechar:D { -1 }

```

```

1124         \cs_set:cpn
1125         {
1126             hook~'#1'. ^^J
1127             (hooks) \prg_replicate:nn { 13 } { ~ }
1128             #2 % more message text
1129         }
1130         \exp_not:v { c__hook_#1_parameter_tl }
1131         {##2}
1132     \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.

```

1133     \cs_set:Npn \exp_not:N \__hook_tmp:w
1134     \exp_not:V \c__hook_nine_parameters_tl
1135     {
1136         \prop_put:Nne \exp_not:N \l__hook_work_prop
1137         {##1} { \exp_not:N \__hook_double_hashes:n {##2} }
1138     }

```

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.

```

1139     \exp_not:N \__hook_tmp:w
1140     \exp_not:V \l__hook_tmpa_tl
1141     \exp_args:No \exp_not:o
1142     { \exp_after:wN \__hook_tmp:w \l__hook_tmpb_tl }
1143 }
1144 }
1145 \cs_new_eq:NN \__hook_normalise_fn:nn ?
1146 <latexrelease>\EndIncludeInRelease
1147 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_normalise_code_pool:n}
1148 <latexrelease>{Hooks~with~args}
1149 <latexrelease>\cs_undefine:N \__hook_normalise_code_pool:n
1150 <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`

```

1151 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_cs_if_empty:c}
1152 <latexrelease>{Hooks~with~args}
1153 \prg_new_conditional:Npnn \__hook_cs_if_empty:c #1 { p, T, F, TF }
1154 {
1155     \if:w \scan_stop: \__hook_replacement_spec:c {#1} \scan_stop:
1156     \prg_return_true:
1157     \else:
1158     \prg_return_false:
1159     \fi:

```

```

1160 }
1161 \cs_new:Npn \__hook_replacement_spec:c #1
1162 {
1163   \exp_args:Nc \token_if_macro:NT {#1}
1164   { \cs_replacement_spec:c {#1} }
1165 }
1166 <latexrelease>\EndIncludeInRelease
1167 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_cs_if_empty:c}
1168 <latexrelease>{Hooks~with~args}
1169 <latexrelease>\cs_undefine:N \__hook_cs_if_empty:c
1170 <latexrelease>\EndIncludeInRelease

```

(End of definition for __hook_normalise_code_pool:n, __hook_set_normalise_fn:nn, and __hook_cs_if_empty:cTF.)

__hook_braced_cs_parameter:n 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.

```

1171 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_braced_cs_parameter:n}
1172 <latexrelease>{Hooks~with~args}
1173 \cs_new:Npn \__hook_braced_cs_parameter:n #1
1174 {
1175   \exp_last_unbraced:Ne \__hook_braced_hidden_loop:w
1176   { \exp_args:Nc \__hook_cs_parameter_count:N {#1} } ? \s__hook_mark
1177 }
1178 \cs_new:Npn \__hook_braced_hidden_loop:w #1
1179 {
1180   \if:w ? #1
1181     \__hook_use_i_delimit_by_s_mark:nw
1182     \fi:
1183     { \exp_not:N \c__hook_hash_tl #1 }
1184     \__hook_braced_hidden_loop:w
1185   }
1186 \cs_new:Npn \__hook_cs_parameter_count:N #1
1187 {
1188   \exp_last_unbraced:Nf \__hook_cs_parameter_count:w
1189   { \token_if_macro:NT #1 { \cs_parameter_spec:N #1 } }
1190   ? \__hook_cs_end:w ? \__hook_cs_end:w ? \__hook_cs_end:w
1191   ? \__hook_cs_end:w ? \__hook_cs_end:w ? \__hook_cs_end:w
1192   ? \__hook_cs_end:w ? \__hook_cs_end:w ? \__hook_cs_end:w
1193   \s__hook_mark
1194 }
1195 \cs_new:Npn \__hook_cs_parameter_count:w #1#2 #3#4 #5#6 #7#8
1196 { #2 #4 #6 #8 \__hook_cs_parameter_count:w }
1197 \cs_new:Npn \__hook_cs_end:w #1 \s__hook_mark { }
1198 <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.

```

1199 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_braced_cs_parameter:n}
1200 <latexrelease>{Hooks~with~args}
1201 <latexrelease>\EndIncludeInRelease

```


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

`_hook_braced_parameter:n` 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.

```

1202 <latexrelease>\IncludeInRelease{2023/06/01}{\_hook_braced_parameter:n}
1203 <latexrelease> {Hooks~with~args}
1204 \cs_new:Npn \_hook_braced_parameter:n #1
1205 {
1206   \if_case:w
1207     \int_eval:n
1208       { \exp_args:Nv \str_count:n { c\_hook_#1_parameter_tl } / 3 }
1209     \exp_stop_f:
1210     \or: {##1}
1211     \or: {##1} {##2}
1212     \or: {##1} {##2} {##3}
1213     \or: {##1} {##2} {##3} {##4}
1214     \or: {##1} {##2} {##3} {##4} {##5}
1215     \or: {##1} {##2} {##3} {##4} {##5} {##6}
1216     \or: {##1} {##2} {##3} {##4} {##5} {##6} {##7}
1217     \or: {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8}
1218     \or: {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8} {##9}
1219     \else:
1220       \msg_expandable_error:nnn { latex2e } { should-not-happen }
1221       { Invalid~parameter~spec. }
1222     \fi:
1223   }
1224 <latexrelease>\EndIncludeInRelease

1225 <latexrelease>\IncludeInRelease{2020/10/01}{\_hook_braced_parameter:n}
1226 <latexrelease> {Hooks~with~args}
1227 <latexrelease>\cs_undefine:N \_hook_braced_parameter:n
1228 <latexrelease>\EndIncludeInRelease

```

(End of definition for `_hook_braced_parameter:n` and `_hook_braced_real_loop:w`.)

`_hook_parameter:n` This is just a shortcut to e- or f-expand to the *<parameter text>* of the hook.

```

1229 <latexrelease>\IncludeInRelease{2023/06/01}{\_hook_parameter:n}
1230 <latexrelease> {Hooks~with~args}
1231 \cs_new:Npn \_hook_parameter:n #1
1232 {
1233   \cs:w c\_hook_
1234   \tl_if_exist:cTF { c\_hook_#1_parameter_tl }
1235     { #1_parameter } { empty }
1236   _tl \cs_end:
1237 }
1238 \cs_new:Npn \_hook_generic_parameter:n #1
1239 { \_hook_generic_parameter:w #1 / / / \s\_hook_mark }
1240 \cs_new:Npn \_hook_generic_parameter:w #1 / #2 / #3 / #4 \s\_hook_mark
1241 {
1242   \cs_if_exist_use:cF { c\_hook_parameter_#1/./#3_tl }
1243     { \c\_hook_empty_tl }
1244 }
1245 <latexrelease>\EndIncludeInRelease

```

```

1246 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_parameter:n}
1247 <latexrelease>{Hooks-with-args}
1248 <latexrelease>\cs_undefine:N \__hook_parameter:n
1249 <latexrelease>\cs_undefine:N \__hook_generic_parameter:n
1250 <latexrelease>\EndIncludeInRelease

```

(End of definition for __hook_parameter:n.)

4.7 Setting rules for hooks code

```

\g__hook_??_code_prop
\__hook~??
\g__hook_??_reversed_tl
\c__hook_??_parameter_tl

```

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

`__hook_` isn’t used, but it has to be defined to trick the code into thinking that `??` is actually a hook.

```

1251 \prop_new:c { g__hook_??_code_prop }
1252 \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.

```

1253 \tl_new:c { g__hook_??_reversed_tl }

```

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

```

1254 <latexrelease>\IncludeInRelease{2023/06/01}{\c__hook_??_parameter_tl}
1255 <latexrelease>{Hooks-with-args}
1256 \tl_const:cn { c__hook_??_parameter_tl } { }
1257 <latexrelease>\EndIncludeInRelease
1258 <latexrelease>\IncludeInRelease{2020/10/01}{\c__hook_??_parameter_tl}
1259 <latexrelease>{Hooks-with-args}
1260 <latexrelease>\cs_undefine:c { c__hook_??_parameter_tl }
1261 <latexrelease>\EndIncludeInRelease

```

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

```

\hook_gset_rule:nnnn
\__hook_gset_rule:nnnn

```

With `\hook_gset_rule:nnnn{<hook>}{<label1>}{<relation>}{<label2>}` a relation is defined between the two code labels for the given `<hook>`. The special hook `??` stands for *any* hook, which sets a default rule (to be used if no other relation between the two hooks exist).

```

1262 \cs_new_protected:Npn \hook_gset_rule:nnnn #1#2#3#4
1263 {
1264   \__hook_normalize_hook_rule_args:Nnnnn \__hook_gset_rule:nnnn
1265   {#1} {#2} {#3} {#4}
1266 }
1267 <latexrelease>\IncludeInRelease{2022/06/01}{\__hook_gset_rule:nnnn}
1268 <latexrelease>{Refuse-setting-rule-for-one-time-hooks}

```

```

1269 \cs_new_protected:Npn \__hook_gset_rule:nnnn #1#2#3#4
1270 {
1271   \__hook_if_deprecated_generic:nT {#1}
1272   {
1273     \__hook_deprecated_generic_warn:n {#1}
1274     \__hook_do_deprecated_generic:Nn \__hook_gset_rule:nnnn {#1}
1275     {#2} {#3} {#4}
1276     \__hook_use_none_delimit_by_s_mark:w
1277   }
1278   \__hook_if_execute_immediately:nT {#1}
1279   {
1280     \msg_error:nnnnnn { hooks } { rule-too-late }
1281     {#1} {#2} {#3} {#4}
1282     \__hook_use_none_delimit_by_s_mark:w
1283   }

```

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

```

1284   \__hook_init_structure:n {#1}

```

Then we clear any previous relationship between both labels.

```

1285   \__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.

```

1286   \cs_if_exist_use:cTF { __hook_rule_#3_gset:nnn }
1287   {
1288     {#1} {#2} {#4}
1289     \__hook_update_hook_code:n {#1}
1290   }
1291   {
1292     \msg_error:nnnnnn { hooks } { unknown-rule }
1293     {#1} {#2} {#3} {#4}
1294   }
1295   \s__hook_mark
1296 }

1297 <latexrelease>\EndIncludeInRelease
1298 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_gset_rule:nnnn}
1299 <latexrelease>{Refuse~setting~rule~for~one~time~hooks}
1300 <latexrelease>\cs_new_protected:Npn \__hook_gset_rule:nnnn #1#2#3#4
1301 <latexrelease> {
1302 <latexrelease>   \__hook_if_deprecated_generic:nT {#1}
1303 <latexrelease>   {
1304 <latexrelease>     \__hook_deprecated_generic_warn:n {#1}
1305 <latexrelease>     \__hook_do_deprecated_generic:Nn \__hook_gset_rule:nnnn {#1}
1306 <latexrelease>     {#2} {#3} {#4}
1307 <latexrelease>     \exp_after:wN \use_none:nnnnnnnnn \use_none:n
1308 <latexrelease>   }
1309 <latexrelease>   \__hook_init_structure:n {#1}
1310 <latexrelease>   \__hook_rule_gclear:nnn {#1} {#2} {#4}
1311 <latexrelease>   \cs_if_exist_use:cTF { __hook_rule_#3_gset:nnn }
1312 <latexrelease>   {
1313 <latexrelease>     {#1} {#2} {#4}
1314 <latexrelease>     \__hook_update_hook_code:n {#1}
1315 <latexrelease>   }
1316 <latexrelease>   {
1317 <latexrelease>     \msg_error:nnnnnn { hooks } { unknown-rule }

```

```

1318 <latexrelease>          {#1} {#2} {#3} {#4}
1319 <latexrelease>          }
1320 <latexrelease>    }
1321 <latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_gset_rule:nnnn` and `__hook_gset_rule:nnnn`. This function is documented on page 16.)

`__hook_rule_before_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.

```

1322 \cs_new_protected:Npn \__hook_rule_before_gset:nnn #1#2#3
1323 {
1324   \__hook_tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _t1 }
1325   { \__hook_label_ordered:nnTF {#2} {#3} { < } { > } }
1326 }
1327 \cs_new_eq:cN { __hook_rule_<_gset:nnn } \__hook_rule_before_gset:nnn
1328 \cs_new_protected:Npn \__hook_rule_after_gset:nnn #1#2#3
1329 {
1330   \__hook_tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#3} {#2} _t1 }
1331   { \__hook_label_ordered:nnTF {#3} {#2} { < } { > } }
1332 }
1333 \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.

```

1334 \cs_new_protected:Npn \__hook_rule_voids_gset:nnn #1#2#3
1335 {
1336   \__hook_tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _t1 }
1337   { \__hook_label_ordered:nnTF {#2} {#3} { -> } { <- } }
1338 }

```

(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
1339 \cs_new_protected:cpn { __hook_rule_incompatible-error_gset:nnn } #1#2#3
1340 { \__hook_tl_gset:cn { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _t1 }
1341   { xE } }
1342 \cs_new_protected:cpn { __hook_rule_incompatible-warning_gset:nnn } #1#2#3
1343 { \__hook_tl_gset:cn { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _t1 }
1344   { xW } }

```

(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 we use `__hook_rule_gclear:nnn` before setting any rule.

```

1345 \cs_new_protected:Npn \__hook_rule_unrelated_gset:nnn #1#2#3 { }
1346 \cs_new_protected:Npn \__hook_rule_gclear:nnn #1#2#3
1347 { \cs_undefine:c { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _t1 } }

```

(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.

```

1348 \cs_new:Npn \__hook_label_pair:nn #1#2
1349 {
1350   \if_case:w \__hook_str_compare:nn {#1} {#2} \exp_stop_f:
1351     #1 | #1 % 0
1352   \or:   #1 | #2 % +1
1353   \else: #2 | #1 % -1
1354   \fi:
1355 }

```

(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_pair:nn`) and if so return true, else return false.

```

\__hook_label_ordered:nnTF
1356 \prg_new_conditional:Nppn \__hook_label_ordered:nn #1#2 { TF }
1357 {
1358   \if_int_compare:w \__hook_str_compare:nn {#1} {#2} > 0 \exp_stop_f:
1359   \prg_return_true:
1360   \else:
1361   \prg_return_false:
1362   \fi:
1363 }

```

(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.

```

1364 \cs_new:Npn \__hook_if_label_case:nnnnn #1#2
1365 {
1366   \cs:w use_
1367   \if_case:w \__hook_str_compare:nn {#1} {#2}
1368     i \or: ii \else: iii \fi: :nnn
1369   \cs_end:
1370 }

```

(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.

```

1371 \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.

```

1372 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_initialize_all:}
1373 <latexrelease>          {Hooks~with~args}
1374 \cs_new_protected:Npn \__hook_initialize_all:
1375 {

```

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.

```

1376 \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.

```

1377 \__hook_debug:n { \prop_gc_clear:N \g__hook_used_prop }
1378 \seq_map_inline:Nn \g__hook_all_seq
1379 {
1380   \tl_if_empty:CTF { c__hook_##1_parameter_tl }
1381   { \__hook_replacing_args_false: }
1382   { \__hook_replacing_args_true: }
1383   \__hook_update_hook_code:n {##1}
1384   \__hook_replacing_args_reset:
1385 }

```

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

```

1386 \__hook_debug:n
1387 {
1388   \iow_term:x { ^^J All~initialized~(non-empty)~hooks: }
1389   \prop_map_inline:Nn \g__hook_used_prop
1390   {
1391     \iow_term:x
1392     { ^^J ~ ##1 ~ -> ~ \cs_replacement_spec:c { __hook-##1 } ~ }
1393   }
1394 }

```

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

```

1395 \__hook_post_initialization_defs:
1396 }

1397 <latexrelease>\EndIncludeInRelease
1398 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_initialize_all:}
1399 <latexrelease>          {Hooks~with~args}
1400 <latexrelease>\cs_gset_protected:Npn \__hook_initialize_all:
1401 <latexrelease> {
1402 <latexrelease>   \cs_gset_eq:NN \__hook_update_hook_code:n \__hook_initialize_hook_code:n
1403 <latexrelease>   \__hook_debug:n { \prop_gc_clear:N \g__hook_used_prop }
1404 <latexrelease>   \seq_map_inline:Nn \g__hook_all_seq
1405 <latexrelease>     { \__hook_update_hook_code:n {##1} }
1406 <latexrelease>   \__hook_debug:n
1407 <latexrelease>   {
1408 <latexrelease>     \iow_term:x{^^JAll~ initialized~ (non-empty)~ hooks:}
1409 <latexrelease>     \prop_map_inline:Nn \g__hook_used_prop

```

```

1410 <latexrelease>          {
1411 <latexrelease>          \iow_term:x
1412 <latexrelease>          { ^^J ~ ##1 ~ -> ~ \cs_replacement_spec:c { __hook~##1 } ~ }
1413 <latexrelease>          }
1414 <latexrelease>      }
1415 <latexrelease>      \cs_gset_eq:NN \hook_use:n \__hook_use_initialized:n
1416 <latexrelease>      \cs_gset_eq:NN \__hook_preamble_hook:n \use_none:n
1417 <latexrelease>  }
1418 <@@=)
1419 <latexrelease>\cs_gset_eq:NN \@expl@@@initialize@all@@
1420 <latexrelease>      \__hook_initialize_all:
1421 <@@=hook>
1422 <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.

```

1423 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_initialize_hook_code:n}
1424 <latexrelease>      {Hooks~with~args}
1425 \cs_new_protected:Npn \__hook_initialize_hook_code:n #1
1426 {
1427   \__hook_debug:n
1428   { \iow_term:x { ^^J Update~code~for~hook~'##1' \on@line :^^J } }

```

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

```

1429   \__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.

```

1430   \__hook_if_usable:nT {#1}
1431   {
1432     \prop_if_empty:cTF { g__hook_#1_code_prop }
1433     {
1434       \__hook_code_gset:ne {#1}
1435     }

```

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.

```

1436       \exp_not:c { __hook_toplevel~#1 } \__hook_braced_parameter:n {#1}
1437       \exp_not:c { __hook_next~#1 } \__hook_braced_parameter:n {#1}
1438     }
1439   }
1440   {

```

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.

```

1441 \__hook_if_reversed:nTF {#1}
1442 { \cs_set_eq:NN \__hook_tl_gput:Nn \__hook_tl_gput_left:Nn
1443 \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_left:NV }
1444 { \cs_set_eq:NN \__hook_tl_gput:Nn \__hook_tl_gput_right:Nn
1445 \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.

```

1446 \prop_set_eq:Nc \l__hook_work_prop { g__hook_#1_code_prop }
1447 \__hook_initialize_single:ccn
1448 { __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.

```

1449 \__hook_debug:n
1450 { \exp_args:NNx \prop_gput:Nnn \g__hook_used_prop {#1} { } }
1451 }
1452 }
1453 }
1454 <latexrelease>\EndIncludeInRelease

1455 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_initialize_hook_code:n}
1456 <latexrelease>{Hooks~with~args}
1457 <latexrelease>\cs_gset_protected:Npn \__hook_initialize_hook_code:n #1
1458 <latexrelease> {
1459 <latexrelease> \__hook_debug:n
1460 <latexrelease> { \iow_term:x { ^^J Update~code~for~hook~'~#1' \on@line :^^J } }
1461 <latexrelease> \__hook_include_legacy_code_chunk:n {#1}
1462 <latexrelease> \__hook_if_usable:nT {#1}
1463 <latexrelease> {
1464 <latexrelease> \prop_if_empty:cTF { g__hook_#1_code_prop }
1465 <latexrelease> {
1466 <latexrelease> \__hook_tl_gset:co { __hook~#1 }
1467 <latexrelease> {
1468 <latexrelease> \cs:w __hook_toplevel~#1 \exp_after:wN \cs_end:
1469 <latexrelease> \cs:w __hook_next~#1 \cs_end:
1470 <latexrelease> }
1471 <latexrelease> }
1472 <latexrelease> {
1473 <latexrelease> \__hook_if_reversed:nTF {#1}
1474 <latexrelease> { \cs_set_eq:NN \__hook_tl_gput:Nn \__hook_tl_gput_left:Nn
1475 <latexrelease> \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_left:NV }
1476 <latexrelease> { \cs_set_eq:NN \__hook_tl_gput:Nn \__hook_tl_gput_right:Nn
1477 <latexrelease> \cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_right:NV }
1478 <latexrelease> \prop_set_eq:Nc \l__hook_work_prop { g__hook_#1_code_prop }
1479 <latexrelease> \__hook_initialize_single:ccn
1480 <latexrelease> { __hook~#1 } { g__hook_#1_labels_clist } {#1}
1481 <latexrelease> \__hook_debug:n
1482 <latexrelease> { \exp_args:NNx \prop_gput:Nnn \g__hook_used_prop {#1} { } }
1483 <latexrelease> }

```



```

1484 <latexrelease>      }
1485 <latexrelease>    }
1486 <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
`_hook_seq_csname:n` character tokens around.

FMi: note to myself: verify

```

1487 \cs_new:Npn \_hook_tl_csname:n #1 { l_hook_label_#1_tl }
1488 \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
`\l_hook_labels_int` given in TAOCP volume 1 pages 263–266. For this algorithm we need a number of local
`\l_hook_front_tl` variables:

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

```

1489 \seq_new:N \l_hook_labels_seq

```

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

```

1490 \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 :

```

1491 \tl_new:N \l_hook_front_tl
1492 \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:

```

1493 \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:NNn` implements the sorting of the code chunks for a
`_hook_initialize_single:ccn` hook and saves the result in the token list for fast execution (#4). The arguments are `<hook-code-plist>`, `<hook-code-tl>`, `<hook-top-level-code-tl>`, `<hook-next-code-tl>`, `<hook-ordered-labels-clist>` and `<hook-name>` (the latter is only used for debugging—the `<hook-rule-plist>` is accessed using the `<hook-name>`).

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

FMi: Needs checking for, just in case ... maybe

```

1494 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_initialize_single:NNn}
1495 <latexrelease>{Hooks-with-args}
1496 \cs_new_protected:Npn \__hook_initialize_single:NNn #1#2#3
1497 {

```

Step T1: Initialize the data structure ...

```

1498 \seq_clear:N \l__hook_labels_seq
1499 \int_zero:N \l__hook_labels_int

```

Store the name of the hook:

```

1500 \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.

```

1501 \prop_map_inline:Nn \l__hook_work_prop
1502 {
1503   \int_incr:N \l__hook_labels_int
1504   \seq_put_right:Nn \l__hook_labels_seq {##1}
1505   \__hook_tl_set:cn { \__hook_tl_csname:n {##1} } { 0 }
1506   \seq_clear_new:c { \__hook_seq_csname:n {##1} }
1507 }

```

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.

```

1508 \prop_map_inline:Nn \l__hook_work_prop
1509 {
1510   \prop_map_inline:Nn \l__hook_work_prop
1511   {
1512     \__hook_if_label_case:nnnnn {##1} {####1}
1513     { \prop_map_break: }
1514     { \__hook_apply_label_pair:nnn {##1} {####1} }
1515     { \__hook_apply_label_pair:nnn {####1} {##1} }
1516     {#3}
1517   }
1518 }

```

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

```

1519 \__hook_debug:n { \__hook_debug_label_data:N \l__hook_work_prop }

```

Step T4:

```

1520 \tl_set:Nn \l__hook_rear_tl { 0 }
1521 \tl_set:cn { \__hook_tl_csname:n { 0 } } { 0 }
1522 \seq_map_inline:Nn \l__hook_labels_seq
1523 {
1524   \int_compare:nNnT { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
1525   {

```

```

1526         \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } }{##1}
1527         \tl_set:Nn \l__hook_rear_tl {##1}
1528     }
1529 }
1530 \tl_set_eq:Nc \l__hook_front_tl { \__hook_tl_csname:n { 0 } }
1531 \__hook_tl_gclear:N #1
1532 \clist_gclear:N #2

```

The whole loop gets combined in steps T5–T7:

```

1533 \bool_while_do:nn { ! \str_if_eq_p:Vn \l__hook_front_tl { 0 } }
1534 {

```

This part is step T5:

```

1535     \int_decr:N \l__hook_labels_int
1536     \prop_get:NVN \l__hook_work_prop \l__hook_front_tl \l__hook_return_tl
1537     \exp_args:NNV \__hook_tl_gput:Nn #1 \l__hook_return_tl
1538     \__hook_clist_gput:NV #2 \l__hook_front_tl
1539     \__hook_debug:n{ \iow_term:x{Handled~ code~ for~ \l__hook_front_tl} }

```

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

```

1540     \seq_map_inline:cn { \__hook_seq_csname:n { \l__hook_front_tl } }
1541     {
1542         \tl_set:cx { \__hook_tl_csname:n {##1} }
1543         { \int_eval:n
1544             { \cs:w \__hook_tl_csname:n {##1} \cs_end: - 1 }
1545         }
1546         \int_compare:nNnT
1547         { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
1548         {
1549             \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } } {##1}
1550             \tl_set:Nn \l__hook_rear_tl {##1}
1551         }
1552     }

```

and here is step T7:

```

1553     \tl_set_eq:Nc \l__hook_front_tl
1554     { \__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.

```

1555     }
1556     \int_compare:nNnF \l__hook_labels_int = 0
1557     {
1558         \iow_term:x{=====}
1559         \iow_term:x{Error:~ label~ rules~ are~ incompatible:}

```

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

...

FMi: improve output on a rainy day

```

1560     \__hook_debug_label_data:N \l__hook_work_prop
1561     \iow_term:x{=====}
1562 }

```

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.

```

1563 \exp_args:NNe \__hook_tl_gput:Nn #1
1564 { \exp_not:c { __hook_toplevel~#3 } \__hook_braced_parameter:n {#3} }
1565 \__hook_tl_gput_right:Ne #1
1566 { \exp_not:c { __hook_next~#3 } \__hook_braced_parameter:n {#3} }
1567 \use:e
1568 {
1569 \cs_gset:cpn { __hook~#3 } \use:c { c__hook_#3_parameter_tl }
1570 { \exp_not:V #1 }
1571 }
1572 }

1573 \cs_generate_variant:Nn \__hook_initialize_single:NNn { cc }
1574 \<latexrelease>\EndIncludeInRelease

1575 \<latexrelease>\IncludeInRelease{2020/10/01}{\__hook_initialize_single:NNn}
1576 \<latexrelease>{Hooks~with~args}
1577 \<latexrelease>\cs_new_protected:Npn \__hook_initialize_single:NNn #1#2#3
1578 \<latexrelease> {
1579 \<latexrelease> \seq_clear:N \l__hook_labels_seq
1580 \<latexrelease> \int_zero:N \l__hook_labels_int
1581 \<latexrelease> \tl_set:Nn \l__hook_cur_hook_tl {#3}
1582 \<latexrelease> \prop_map_inline:Nn \l__hook_work_prop
1583 \<latexrelease> {
1584 \<latexrelease> \int_incr:N \l__hook_labels_int
1585 \<latexrelease> \seq_put_right:Nn \l__hook_labels_seq {##1}
1586 \<latexrelease> \__hook_tl_set:cn { \__hook_tl_csname:n {##1} } { 0 }
1587 \<latexrelease> \seq_clear_new:c { \__hook_seq_csname:n {##1} }
1588 \<latexrelease> }
1589 \<latexrelease> \prop_map_inline:Nn \l__hook_work_prop
1590 \<latexrelease> {
1591 \<latexrelease> \prop_map_inline:Nn \l__hook_work_prop
1592 \<latexrelease> {
1593 \<latexrelease> \__hook_if_label_case:nnnn {##1} {####1}
1594 \<latexrelease> { \prop_map_break: }
1595 \<latexrelease> { \__hook_apply_label_pair:nnn {##1} {####1} }
1596 \<latexrelease> { \__hook_apply_label_pair:nnn {####1} {##1} }
1597 \<latexrelease> {#3}
1598 \<latexrelease> }
1599 \<latexrelease> }
1600 \<latexrelease> \__hook_debug:n { \__hook_debug_label_data:N \l__hook_work_prop }
1601 \<latexrelease> \tl_set:Nn \l__hook_rear_tl { 0 }
1602 \<latexrelease> \tl_set:cn { \__hook_tl_csname:n { 0 } } { 0 }
1603 \<latexrelease> \seq_map_inline:Nn \l__hook_labels_seq
1604 \<latexrelease> {
1605 \<latexrelease> \int_compare:nNnT { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
1606 \<latexrelease> {
1607 \<latexrelease> \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } } {##1}
1608 \<latexrelease> \tl_set:Nn \l__hook_rear_tl {##1}

```

```

1609 <latexrelease>      }
1610 <latexrelease>      }
1611 <latexrelease>      \tl_set_eq:Nc \l__hook_front_tl { \__hook_tl_csname:n { 0 } }
1612 <latexrelease>      \__hook_tl_gclear:N #1
1613 <latexrelease>      \clist_gclear:N #2
1614 <latexrelease>      \bool_while_do:nn { ! \str_if_eq_p:Vn \l__hook_front_tl { 0 } }
1615 <latexrelease>      {
1616 <latexrelease>          \int_decr:N \l__hook_labels_int
1617 <latexrelease>          \prop_get:NVN \l__hook_work_prop \l__hook_front_tl \l__hook_return_tl
1618 <latexrelease>          \exp_args:NNV \__hook_tl_gput:Nn #1 \l__hook_return_tl
1619 <latexrelease>          \__hook_clist_gput:NV #2 \l__hook_front_tl
1620 <latexrelease>          \__hook_debug:n{ \iow_term:x{Handled~ code~ for~ \l__hook_front_tl} }
1621 <latexrelease>          \seq_map_inline:cn { \__hook_seq_csname:n { \l__hook_front_tl } }
1622 <latexrelease>          {
1623 <latexrelease>              \tl_set:cx { \__hook_tl_csname:n {##1} }
1624 <latexrelease>              { \int_eval:n
1625 <latexrelease>                  { \cs:w \__hook_tl_csname:n {##1} \cs_end: - 1 }
1626 <latexrelease>              }
1627 <latexrelease>          \int_compare:nNnT
1628 <latexrelease>              { \cs:w \__hook_tl_csname:n {##1} \cs_end: } = 0
1629 <latexrelease>          {
1630 <latexrelease>              \tl_set:cn { \__hook_tl_csname:n { \l__hook_rear_tl } } {##1}
1631 <latexrelease>              \tl_set:Nn \l__hook_rear_tl {##1}
1632 <latexrelease>          }
1633 <latexrelease>      }
1634 <latexrelease>      \tl_set_eq:Nc \l__hook_front_tl
1635 <latexrelease>          { \__hook_tl_csname:n { \l__hook_front_tl } }
1636 <latexrelease>      }
1637 <latexrelease>      \int_compare:nNnF \l__hook_labels_int = 0
1638 <latexrelease>      {
1639 <latexrelease>          \iow_term:x{=====}
1640 <latexrelease>          \iow_term:x{Error:~ label~ rules~ are~ incompatible:}
1641 <latexrelease>          \__hook_debug_label_data:N \l__hook_work_prop
1642 <latexrelease>          \iow_term:x{=====}
1643 <latexrelease>      }
1644 <latexrelease>      \exp_args:NNo \__hook_tl_gput:Nn #1 { \cs:w __hook_toplevel~#3 \cs_end: }
1645 <latexrelease>      \__hook_tl_gput_right:No #1 { \cs:w __hook_next~#3 \cs_end: }
1646 <latexrelease>      }
1647 <latexrelease> \cs_generate_variant:Nn \__hook_tl_gput_right:Nn { No }
1648 <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.

```

1649 \cs_new:Npn \__hook_tl_gput:Nn    { \ERROR }
1650 \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:nnmF 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 $\langle label1 \rangle$, $\langle label2 \rangle$, $\langle hook \rangle$, and $\langle hook-code-plist \rangle$. 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 $\langle hook \rangle$, and if no rule is found, also try the $\langle hook \rangle$ 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`.

```
1651 \cs_new_protected:Npn \_hook_apply_label_pair:nnn #1#2#3
1652 {
```

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.

```
1653   \_hook_label_if_exist_apply:nnnF {#1} {#2} {#3}
1654   {
```

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

```
1655       \_hook_label_if_exist_apply:nnnF {#1} {#2} { ?? } { }
1656   }
1657 }
1658 \cs_new_protected:Npn \_hook_label_if_exist_apply:nnnF #1#2#3
1659 {
1660   \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`.

```
1661   \_hook_apply_rule:nnn {#1} {#2} {#3}
1662   \exp_after:wN \use_none:n
1663   \else:
1664     \use:nn
1665   \fi:
1666 }
```

(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 $\langle hook-code-plist \rangle$.

```
1667 \cs_new_protected:Npn \_hook_apply_rule:nnn #1#2#3
1668 {
1669   \cs:w __hook_apply_
1670   \cs:w g__hook_#3_reversed_tl \cs_end: rule_
1671   \cs:w g__hook_ #3 _rule_ #1 | #2 _tl \cs_end: :nnn \cs_end:
1672   {#1} {#2} {#3}
1673 }
```

(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 \prec and \succ in TAOCP, and they dictate sorting.

```
\_hook_apply_rule_>:nnn
1674 \cs_new_protected:cpn { __hook_apply_rule_<:nnn } #1#2#3
1675 {
1676   \_hook_debug:n { \_hook_msg_pair_found:nnn {#1} {#2} {#3} }
1677   \tl_set:cx { \_hook_tl_csname:n {#2} }
```

```

1678         { \int_eval:n{ \cs:w \__hook_tl_csname:n {#2} \cs_end: + 1 } }
1679         \seq_put_right:cn{ \__hook_seq_csname:n {#1} }{#2}
1680     }
1681     \cs_new_protected:cpn { __hook_apply_rule_>:nnn } #1#2#3
1682     {
1683         \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
1684         \tl_set:cx { \__hook_tl_csname:n {#1} }
1685         { \int_eval:n{ \cs:w \__hook_tl_csname:n {#1} \cs_end: + 1 } }
1686         \seq_put_right:cn{ \__hook_seq_csname:n {#2} }{#1}
1687     }

```

(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.

```

1688 \cs_new_protected:cpn { __hook_apply_rule_xE:nnn } #1#2#3
1689 {
1690     \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
1691     \msg_error:nnnnnn { hooks } { labels-incompatible }
1692     {#1} {#2} {#3} { 1 }
1693     \use:c { __hook_apply_rule_>:nnn } {#1} {#2} {#3}
1694     \use:c { __hook_apply_rule_<:nnn } {#1} {#2} {#3}
1695 }
1696 \cs_new_protected:cpn { __hook_apply_rule_xW:nnn } #1#2#3
1697 {
1698     \__hook_debug:n { \__hook_msg_pair_found:nnn {#1} {#2} {#3} }
1699     \msg_warning:nnnnnn { hooks } { labels-incompatible }
1700     {#1} {#2} {#3} { 0 }
1701 }

```

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

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

```

1702 \cs_new_protected:cpn { __hook_apply_rule_>:nnn } #1#2#3
1703 {
1704     \__hook_debug:n
1705     {
1706         \__hook_msg_pair_found:nnn {#1} {#2} {#3}
1707         \iow_term:x{--->~ Drop~ '#2'~ code~ from~
1708         \iow_char:N \ g__hook_ \l__hook_cur_hook_tl _code_prop ~
1709         because~ of~ '#1' }
1710     }
1711     \prop_put:Nnn \l__hook_work_prop {#2} { }
1712 }

```

⁹This 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.

```

1713 \cs_new_protected:cpn { __hook_apply_rule_<:nnn } #1#2#3
1714 {
1715   \__hook_debug:n
1716   {
1717     \__hook_msg_pair_found:nnn {#1} {#2} {#3}
1718     \iow_term:x{--->~ Drop~ '1'~ code~ from~
1719       \iow_char:N \ \ g__hook_ \l__hook_cur_hook_tl _code_prop ~
1720       because~ of~ '2' }
1721   }
1722   \prop_put:Nnn \l__hook_work_prop {#1} { }
1723 }

```

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

Reversed rules.

```

\__hook_apply_rule_<:nnn
\__hook_apply_rule_>:nnn
\__hook_apply_rule_<:nnn
\__hook_apply_rule_>:nnn
\__hook_apply_rule_xW:nnn
\__hook_apply_rule_xE:nnn
1724 \cs_new_eq:cc { __hook_apply_rule_<:nnn } { __hook_apply_rule_>:nnn }
1725 \cs_new_eq:cc { __hook_apply_rule_>:nnn } { __hook_apply_rule_<:nnn }
1726 \cs_new_eq:cc { __hook_apply_rule_<:nnn } { __hook_apply_rule_>:nnn }
1727 \cs_new_eq:cc { __hook_apply_rule_xW:nnn } { __hook_apply_rule_xE:nnn }
1728 \cs_new_eq:cc { __hook_apply_rule_xE:nnn } { __hook_apply_rule_xW:nnn }
1729 \cs_new_eq:cc { __hook_apply_rule_xW:nnn } { __hook_apply_rule_xE: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.

```

1730 \cs_new_protected:Npn \__hook_msg_pair_found:nnn #1#2#3
1731 {
1732   \iow_term:x{~ \str_if_eq:nnTF {#3} {??} {default} {~normal} ~
1733     rule~ \__hook_label_pair:nn {#1} {#2}:~
1734     \use:c { g__hook_#3_rule_ \__hook_label_pair:nn {#1} {#2} _tl } ~
1735     found}
1736 }

```

(End of definition for __hook_msg_pair_found:nnn.)

__hook_debug_label_data:N

```

1737 \cs_new_protected:Npn \__hook_debug_label_data:N #1 {
1738   \iow_term:x{Code~ labels~ for~ sorting:}
1739   \iow_term:x{~ \seq_use:Nnnn\l__hook_labels_seq {~and~}{,~}{~and~} }
1740   \iow_term:x{^^J Data~ structure~ for~ label~ rules:}
1741   \prop_map_inline:Nn #1
1742   {
1743     \iow_term:x{~ ##1~ =~ \tl_use:c{ \__hook_tl_csname:n {##1} }~ ->~
1744       \seq_use:cnnn{ \__hook_seq_csname:n {##1} }{~>~}{~>~}{~>~}
1745   }
1746 }
1747 \iow_term:x{}
1748 }

```

(End of definition for __hook_debug_label_data:N.)

\hook_show:n

\hook_log:n

__hook_log_line:x

__hook_log_line_indent:x

__hook_log:nN

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

```

1749 \cs_new_protected:Npn \hook_log:n #1

```



```

1750 {
1751   \cs_set_eq:NN \__hook_log_cmd:x \iow_log:x
1752   \__hook_normalize_hook_args:Nn \__hook_log:nN {#1} \tl_log:x
1753 }
1754 \cs_new_protected:Npn \hook_show:n #1
1755 {
1756   \cs_set_eq:NN \__hook_log_cmd:x \iow_term:x
1757   \__hook_normalize_hook_args:Nn \__hook_log:nN {#1} \tl_show:x
1758 }
1759 \cs_new_protected:Npn \__hook_log_line:x #1
1760 { \__hook_log_cmd:x { >~#1 } }
1761 \cs_new_protected:Npn \__hook_log_line_indent:x #1
1762 { \__hook_log_cmd:x { >~\@spaces #1 } }

1763 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_log:nN}
1764 <latexrelease>{Hooks-with-args}
1765 \cs_new_protected:Npn \__hook_log:nN #1 #2
1766 {
1767   \__hook_if_deprecated_generic:nT {#1}
1768   {
1769     \__hook_deprecated_generic_warn:n {#1}
1770     \__hook_do_deprecated_generic:Nn \__hook_log:nN {#1} #2
1771     \exp_after:wN \use_none:nnnnnnnn \use_none:nnnnn
1772   }
1773   \__hook_preamble_hook:n {#1}
1774   \__hook_log_cmd:x
1775   {
1776     ^^J ->~The~
1777     \__hook_if_generic:nT {#1} { generic~ }
1778     hook~'~#1'
1779     \__hook_if_disabled:nF {#1}
1780     {
1781       \exp_args:Nf \__hook_print_args:nn {#1}
1782       {
1783         \int_eval:n
1784         { \str_count:e { \__hook_parameter:n {#1} } / 3 }
1785       }
1786     }
1787     :
1788   }

1789   \__hook_if_usable:nF {#1}
1790   { \__hook_log_line:x { The~hook~is~not~declared. } }
1791   \__hook_if_disabled:nT {#1}
1792   { \__hook_log_line:x { The~hook~is~disabled. } }
1793   \hook_if_empty:nTF {#1}
1794   { #2 { The~hook~is~empty } }
1795   {
1796     \__hook_log_line:x { Code~chunks: }
1797     \prop_if_empty:cTF { g__hook_#1_code_prop }
1798     { \__hook_log_line_indent:x { --- } }
1799     {
1800       \prop_map_inline:cn { g__hook_#1_code_prop }
1801       {
1802         \exp_after:wN \cs_set:Npn \exp_after:wN \__hook_tmp:w

```

```

1803         \c_hook_nine_parameters_t1 {##2}
1804         \__hook_log_line_indent:x
1805         { ##1~-->~\cs_replacement_spec:N \__hook_tmp:w }
1806     }
1807 }

```

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

```

1808     \__hook_log_line:x
1809     {
1810         Document-level~(top-level)~code
1811         \__hook_if_usable:nT {#1}
1812         { ~(executed~\__hook_if_reversed:nTF {#1} {first} {last} ) } :
1813     }
1814     \__hook_log_line_indent:x
1815     {
1816         \__hook_cs_if_empty:cTF { __hook_toplevel~#1 }
1817         { --- }
1818         { -> ~ \cs_replacement_spec:c { __hook_toplevel~#1 } }
1819     }
1820     \__hook_log_line:x { Extra~code~for~next~invocation: }
1821     \__hook_log_line_indent:x
1822     {
1823         \__hook_cs_if_empty:cTF { __hook_next~#1 }
1824         { --- }

```

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.

```

1825     {
1826         -> ~ \exp_last_unbraced:Nf \__hook_log_next_code:w
1827         { \cs_replacement_spec:c { __hook_next~#1 } }
1828     }
1829 }

```

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.

```

1830     \__hook_log_line:x { Rules: }
1831     \bool_set_true:N \l__hook_tmpa_bool
1832     \__hook_list_rules:nn {#1}
1833     {
1834         \bool_set_false:N \l__hook_tmpa_bool
1835         \__hook_log_line_indent:x
1836         {
1837             ##2~ with~
1838             \str_if_eq:nnT {##3} {??} { default~ }
1839             relation~ ##1
1840         }
1841     }
1842     \bool_if:NT \l__hook_tmpa_bool
1843     { \__hook_log_line_indent:x { --- } }

```

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

```

1844     \bool_lazy_and:nnTF

```

```

1845 { \_hook_if_usable_p:n {#1} }
1846 { ! \hook_if_empty_p:n {#1} }
1847 {
1848   \_hook_log_line:x
1849   {
1850     Execution~order
1851     \bool_if:NTF \l__hook_tmpa_bool
1852     { \_hook_if_reversed:nT {#1} { ~(after~reversal) } }
1853     { ~(after~
1854       \_hook_if_reversed:nT {#1} { reversal~and~
1855       applying~rules)
1856     } :
1857   }
1858   #2 % \tl_show:n
1859   {
1860     \@spaces
1861     \clist_if_empty:cTF { g__hook_#1_labels_clist }
1862     { --- }
1863     { \clist_use:cn { g__hook_#1_labels_clist } { ,~ } }
1864   }
1865 }
1866 {
1867   \_hook_log_line:x { Execution~order: }
1868   #2
1869   {
1870     \@spaces Not~set~because~the~hook~ \_hook_if_usable:nTF {#1}
1871     { code~pool~is~empty }
1872     { is~\_hook_if_disabled:nTF {#1} {disabled} {undeclared} }
1873   }
1874 }
1875 }
1876 }
1877 <latexrelease>\EndIncludeInRelease
1878 %
1879 <latexrelease>\IncludeInRelease{2020/10/01}{\_hook_log:nN}
1880 <latexrelease> {Hooks~with~args}
1881 <latexrelease>\cs_new_protected:Npn \_hook_log:nN #1 #2
1882 <latexrelease> {
1883 <latexrelease>   \_hook_if_deprecated_generic:nT {#1}
1884 <latexrelease>   {
1885 <latexrelease>     \_hook_deprecated_generic_warn:n {#1}
1886 <latexrelease>     \_hook_do_deprecated_generic:Nn \_hook_log:nN {#1} #2
1887 <latexrelease>     \exp_after:wN \use_none:nnnnnnnnn \use_none:nnnnn
1888 <latexrelease>   }
1889 <latexrelease>   \_hook_preamble_hook:n {#1}
1890 <latexrelease>   \_hook_log_cmd:x
1891 <latexrelease>   { ^~J ->~The~ \_hook_if_generic:nT {#1} { generic~ } hook~'#1': }
1892 <latexrelease>   \_hook_if_usable:nF {#1}
1893 <latexrelease>   { \_hook_log_line:x { The~hook~is~not~declared. } }
1894 <latexrelease>   \_hook_if_disabled:nT {#1}
1895 <latexrelease>   { \_hook_log_line:x { The~hook~is~disabled. } }
1896 <latexrelease>   \hook_if_empty:nTF {#1}
1897 <latexrelease>   { #2 { The~hook~is~empty } }
1898 <latexrelease>   {

```

```

1899 <latexrelease> \_hook_log_line:x { Code~chunks: }
1900 <latexrelease> \prop_if_empty:cTF { g__hook_#1_code_prop }
1901 <latexrelease> { \_hook_log_line_indent:x { --- } }
1902 <latexrelease> {
1903 <latexrelease> \prop_map_inline:cn { g__hook_#1_code_prop }
1904 <latexrelease> { \_hook_log_line_indent:x { ##1~>~\tl_to_str:n {##2} } }
1905 <latexrelease> }
1906 <latexrelease> \_hook_log_line:x
1907 <latexrelease> {
1908 <latexrelease> Document-level~(top-level)~code
1909 <latexrelease> \_hook_if_usable:nT {#1}
1910 <latexrelease> { ~(executed~\_hook_if_reversed:nTF {#1} {first} {last} ) } :
1911 <latexrelease> }
1912 <latexrelease> \_hook_log_line_indent:x
1913 <latexrelease> {
1914 <latexrelease> \tl_if_empty:cTF { __hook_toplevel~#1 }
1915 <latexrelease> { --- }
1916 <latexrelease> { -> ~ \exp_args:Nv \tl_to_str:n { __hook_toplevel~#1 } }
1917 <latexrelease> }
1918 <latexrelease> \_hook_log_line:x { Extra~code~for~next~invocation: }
1919 <latexrelease> \_hook_log_line_indent:x
1920 <latexrelease> {
1921 <latexrelease> \tl_if_empty:cTF { __hook_next~#1 }
1922 <latexrelease> { --- }
1923 <latexrelease> { ->~ \exp_args:Nv \_hook_log_next_code:n { __hook_next~#1 } }
1924 <latexrelease> }
1925 <latexrelease> \_hook_log_line:x { Rules: }
1926 <latexrelease> \bool_set_true:N \l__hook_tmpa_bool
1927 <latexrelease> \_hook_list_rules:nn {#1}
1928 <latexrelease> {
1929 <latexrelease> \bool_set_false:N \l__hook_tmpa_bool
1930 <latexrelease> \_hook_log_line_indent:x
1931 <latexrelease> {
1932 <latexrelease> ##2~ with~
1933 <latexrelease> \str_if_eq:nnT {##3} {??} { default~ }
1934 <latexrelease> relation~ ##1
1935 <latexrelease> }
1936 <latexrelease> }
1937 <latexrelease> \bool_if:NT \l__hook_tmpa_bool
1938 <latexrelease> { \_hook_log_line_indent:x { --- } }
1939 <latexrelease> \bool_lazy_and:nnTF
1940 <latexrelease> { \_hook_if_usable_p:n {#1} }
1941 <latexrelease> { ! \hook_if_empty_p:n {#1} }
1942 <latexrelease> {
1943 <latexrelease> \_hook_log_line:x
1944 <latexrelease> {
1945 <latexrelease> Execution~order
1946 <latexrelease> \bool_if:NTF \l__hook_tmpa_bool
1947 <latexrelease> { \_hook_if_reversed:nT {#1} { ~(after~reversal) } }
1948 <latexrelease> { ~(after~
1949 <latexrelease> \_hook_if_reversed:nT {#1} { reversal~and~ }
1950 <latexrelease> applying~rules)
1951 <latexrelease> } :
1952 <latexrelease> }

```

```

1953 <latexrelease>          #2 % \tl_show:n
1954 <latexrelease>          {
1955 <latexrelease>            \@spaces
1956 <latexrelease>            \clist_if_empty:cTF { g__hook_#1_labels_clist }
1957 <latexrelease>            { --- }
1958 <latexrelease>            { \clist_use:cn { g__hook_#1_labels_clist } { ,~ } }
1959 <latexrelease>          }
1960 <latexrelease>        }
1961 <latexrelease>        {
1962 <latexrelease>          \__hook_log_line:x { Execution~order: }
1963 <latexrelease>          #2
1964 <latexrelease>          {
1965 <latexrelease>            \@spaces Not~set~because~the~hook~ \__hook_if_usable:nTF {#1}
1966 <latexrelease>            { code~pool~is~empty }
1967 <latexrelease>            { is~\__hook_if_disabled:nTF {#1} {disabled} {undeclared} }
1968 <latexrelease>          }
1969 <latexrelease>        }
1970 <latexrelease>      }
1971 <latexrelease>    }
1972 <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.

```

1973 <latexrelease> \IncludeInRelease{2023/06/01}{\__hook_log_next_code:n}
1974 <latexrelease>          {Hooks~with~args}
__hook_log_next_code:n
1975 \exp_last_unbraced:NNNNo
1976 \cs_new:Npn \__hook_log_next_code:w #1 \c_right_brace_str { }
1977 <latexrelease> \EndIncludeInRelease
1978 <latexrelease> \IncludeInRelease{2020/10/01}{\__hook_log_next_code:n}
1979 <latexrelease>          {Hooks~with~args}
1980 <latexrelease> \cs_gset:Npn \__hook_log_next_code:n #1
1981 <latexrelease>   { \exp_args:No \tl_to_str:n { \use_none:nn #1 } }
1982 <latexrelease> \EndIncludeInRelease

```

Pretty-prints the number of arguments of a hook.

```

1983 \cs_new:Npn \__hook_print_args:nn #1 #2
1984 {
1985   \int_compare:nNnT {#2} > { 0 }
1986   {
__hook_print_args:n
1987     \__hook_if_declared:nT {#1} { \use_none:nnn }
1988     \__hook_if_cmd_hook:nT {#1}
1989     { \use_i:nnn { ~ (unknown ~ } }
1990     \use:n { ~ (#2 ~ }
1991     argument \int_compare:nNnT {#2} > { 1 } { s } )
1992   }
1993 }

```

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

```

__hook_list_rules:nn
__hook_list_one_rule:nnn
__hook_list_if_rule_exists:nnnF

```

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

```

1994 \cs_new_protected:Npn \__hook_list_rules:nn #1 #2
1995 {
1996   \cs_set_protected:Npn \__hook_tmp:w ##1 ##2 ##3 {#2}
1997   \prop_map_inline:cn { g__hook_#1_code_prop }
1998   {
1999     \prop_map_inline:cn { g__hook_#1_code_prop }
2000     {
2001       \__hook_if_label_case:nnnnn {##1} {####1}
2002       { \prop_map_break: }
2003       { \__hook_list_one_rule:nnn {##1} {####1} }
2004       { \__hook_list_one_rule:nnn {####1} {##1} }
2005       {#1}
2006     }
2007   }
2008 }

```

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*.

```

2009 \cs_new_protected:Npn \__hook_list_one_rule:nnn #1#2#3
2010 {
2011   \__hook_list_if_rule_exists:nnnF {#1} {#2} {#3}
2012   { \__hook_list_if_rule_exists:nnnF {#1} {#2} { ?? } { } }
2013 }
2014 \cs_new_protected:Npn \__hook_list_if_rule_exists:nnnF #1#2#3
2015 {
2016   \if_cs_exist:w g__hook_#3_rule_#1 | #2 _t1 \cs_end:
2017   \exp_args:Nv \__hook_tmp:w
2018   { g__hook_#3_rule_#1 | #2 _t1 } { #1 | #2 } {#3}
2019   \exp_after:wN \use_none:nn
2020   \fi:
2021   \use:n
2022 }

```

(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`.

```

2023 \cs_new_protected:Npn \__hook_debug_print_rules:n #1
2024 {
2025   \iow_term:n { The~hook~#1~contains~the~rules: }
2026   \cs_set_protected:Npn \__hook_tmp:w ##1
2027   {
2028     \__hook_list_rules:nn {#1}
2029     {
2030       \iow_term:x
2031       {
2032         > ##1 {####2} ##1 => ##1 {####1}
2033         \str_if_eq:nnT {####3} {??} { ~(default) }
2034       }
2035     }
2036   }
2037   \exp_args:No \__hook_tmp:w { \use:nn { ~ } { ~ } }
2038 }

```

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

4.8 Specifying code for next invocation

`\hook_gput_next_code:nn`

```

2039 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_gput_next_code:nn}
2040 <latexrelease> {Hooks~with~args}
2041 \cs_new_protected:Npn \hook_gput_next_code:nn #1 #2
2042 {
2043   \__hook_replacing_args_false:
2044   \__hook_normalize_hook_args:Nn \__hook_gput_next_code:nn {#1} {#2}
2045   \__hook_replacing_args_reset:
2046 }
2047 \cs_new_protected:Npn \hook_gput_next_code_with_args:nn #1 #2
2048 {
2049   \__hook_replacing_args_true:
2050   \__hook_normalize_hook_args:Nn \__hook_gput_next_code:nn {#1} {#2}
2051   \__hook_replacing_args_reset:
2052 }
2053 <latexrelease>\EndIncludeInRelease
2054 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_gput_next_code:nn}
2055 <latexrelease> {Hooks~with~args}
2056 <latexrelease>\cs_gset_protected:Npn \hook_gput_next_code:nn #1
2057 <latexrelease> { \__hook_normalize_hook_args:Nn \__hook_gput_next_code:nn {#1} }
2058 <latexrelease>\cs_gset_protected:Npn \hook_gput_next_code_with_args:nn #1 #2 { }
2059 <latexrelease>\EndIncludeInRelease

```

(End of definition for `\hook_gput_next_code:nn`. This function is documented on page 15.)

`__hook_gput_next_code:nn`

```

2060 \cs_new_protected:Npn \__hook_gput_next_code:nn #1 #2
2061 {
2062   \__hook_if_disabled:nTF {#1}
2063   { \msg_error:nnn { hooks } { hook-disabled } {#1} }
2064   {
2065     \__hook_if_structure_exist:nTF {#1}
2066     { \__hook_gput_next_do:nn }
2067     { \__hook_try_declaring_generic_next_hook:nn }
2068     {#1} {#2}
2069   }
2070 }

```

(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.

```

2071 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_gput_next_do:nn}
2072 <latexrelease> {Hooks~with~args}
2073 \cs_new_protected:Npn \__hook_gput_next_do:nn #1
2074 {
2075   \__hook_init_structure:n {#1}

```

```

2076 \__hook_chk_args_allowed:nn {#1} { AddToHookNext }
2077 \__hook_cs_if_empty:cT { __hook~#1 }
2078 { \__hook_update_hook_code:n {#1} }
2079 \__hook_cs_if_empty:cT { __hook_next~#1 }
2080 { \__hook_next_gset:nn {#1} { \__hook_clear_next:n {#1} } }
2081 \__hook_cs_gput_right:nnn { _next } {#1}
2082 }
2083 <latexrelease>\EndIncludeInRelease
2084 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_gput_next_do:nn}
2085 <latexrelease> {Hooks~with~args}
2086 <latexrelease>\cs_gset_protected:Npn \__hook_gput_next_do:nn #1
2087 <latexrelease> {
2088 <latexrelease> \exp_args:Nc \__hook_gput_next_do:Nnn
2089 <latexrelease> { __hook_next~#1 } {#1}
2090 <latexrelease> }
2091 <latexrelease>\cs_gset_protected:Npn \__hook_gput_next_do:Nnn #1 #2
2092 <latexrelease> {
2093 <latexrelease> \tl_if_empty:cT { __hook~#2 }
2094 <latexrelease> { \__hook_update_hook_code:n {#2} }
2095 <latexrelease> \tl_if_empty:NT #1
2096 <latexrelease> { \__hook_tl_gset:Nn #1 { \__hook_clear_next:n {#2} } }
2097 <latexrelease> \__hook_tl_gput_right:Nn #1
2098 <latexrelease> }
2099 <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.

```

2100 \cs_new_protected:Npn \hook_gclear_next_code:n #1
2101 { \__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 16.)

__hook_clear_next:n

```

2102 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_clear_next:n}
2103 <latexrelease> {Hooks~with~args}
2104 \cs_new_protected:Npn \__hook_clear_next:n #1
2105 { \__hook_next_gset:nn {#1} { } }
2106 <latexrelease>\EndIncludeInRelease
2107 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_clear_next:n}
2108 <latexrelease> {Hooks~with~args}
2109 <latexrelease>\cs_gset_protected:Npn \__hook_clear_next:n #1
2110 <latexrelease> { \cs_gset_eq:cN { __hook_next~#1 } \c_empty_tl }
2111 <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.

```

2112 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_use:n}
2113 <latexrelease> {Hooks-with-args}
2114 \cs_new_protected:Npn \hook_use:n #1
2115 {
2116   \__hook_preamble_hook:n {#1}
2117   \__hook_use_initialized:n {#1}
2118 }
2119 \cs_new:Npn \__hook_use_initialized:n #1
2120 {
2121   \if_cs_exist:w __hook~#1 \cs_end:
2122   \cs:w __hook~#1 \use_i:nn
2123   \fi:
2124   \use_none:n
2125   \cs_end:
2126 }
2127 \cs_new_protected:Npn \__hook_preamble_hook:n #1
2128 {
2129   \if_cs_exist:w __hook~#1 \cs_end:
2130   \__hook_initialize_hook_code:n {#1}
2131   \fi:
2132 }
2133 <latexrelease>\EndIncludeInRelease
2134 <latexrelease>\IncludeInRelease{2021/11/15}{\hook_use:n}
2135 <latexrelease> {Standardise-generic-hook-names}
2136 <latexrelease>\cs_new_protected:Npn \hook_use:n #1
2137 <latexrelease> {
2138 <latexrelease>   \tl_if_exist:cT { __hook~#1 }
2139 <latexrelease>   {
2140 <latexrelease>     \__hook_preamble_hook:n {#1}
2141 <latexrelease>     \cs:w __hook~#1 \cs_end:
2142 <latexrelease>   }
2143 <latexrelease> }
2144 <latexrelease>\cs_new:Npn \__hook_use_initialized:n #1
2145 <latexrelease> {
2146 <latexrelease>   \if_cs_exist:w __hook~#1 \cs_end:
2147 <latexrelease>   \cs:w __hook~#1 \exp_after:wN \cs_end:
2148 <latexrelease>   \fi:
2149 <latexrelease> }
2150 <latexrelease>\cs_new_protected:Npn \__hook_preamble_hook:n #1
2151 <latexrelease> { \__hook_initialize_hook_code:n {#1} }
2152 <latexrelease>\cs_new:Npn \hook_use:nnw #1 { }
2153 <latexrelease>\EndIncludeInRelease

```

```

2154 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_use:n}
2155 <latexrelease> {Standardise-generic-hook-names}
2156 <latexrelease>\cs_new_protected:Npn \hook_use:n #1
2157 <latexrelease> {
2158 <latexrelease> \tl_if_exist:cTF { __hook~#1 }
2159 <latexrelease> {
2160 <latexrelease> \__hook_preamble_hook:n {#1}
2161 <latexrelease> \cs:w __hook~#1 \cs_end:
2162 <latexrelease> }
2163 <latexrelease> { \__hook_use:wn #1 / \s__hook_mark {#1} }
2164 <latexrelease> }
2165 <latexrelease>\cs_new:Npn \__hook_use_initialized:n #1
2166 <latexrelease> {
2167 <latexrelease> \if_cs_exist:w __hook~#1 \cs_end:
2168 <latexrelease> \else:
2169 <latexrelease> \__hook_use_undefined:w
2170 <latexrelease> \fi:
2171 <latexrelease> \cs:w __hook~#1 \__hook_use_end:
2172 <latexrelease> }
2173 <latexrelease>\cs_new:Npn \__hook_use_undefined:w #1 #2 __hook~#3 \__hook_use_end:
2174 <latexrelease> {
2175 <latexrelease> #1 % fi
2176 <latexrelease> \__hook_use:wn #3 / \s__hook_mark {#3}
2177 <latexrelease> }
2178 <latexrelease>\cs_new_protected:Npn \__hook_preamble_hook:n #1
2179 <latexrelease> { \__hook_initialize_hook_code:n {#1} }
2180 <latexrelease>\cs_new_eq:NN \__hook_use_end: \cs_end:
2181 <latexrelease>\cs_new:Npn \hook_use:nnw #1 { }
2182 <latexrelease>\EndIncludeInRelease

```

(End of definition for \hook_use:n, __hook_use_initialized:n, and __hook_preamble_hook:n. This function is documented on page 15.)

\hook_use:nnw

__hook_use_initialized:nnw

```

2183 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_use:nnw}
2184 <latexrelease> {Hooks~with~args}
2185 \cs_new_protected:Npn \hook_use:nnw #1
2186 {
2187 \__hook_preamble_hook:n {#1}
2188 \__hook_use_initialized:nnw {#1}
2189 }
2190 \cs_new:Npn \__hook_use_initialized:nnw #1 #2
2191 {
2192 \cs:w
2193 \if_cs_exist:w __hook~#1 \cs_end:
2194 __hook~#1
2195 \else:
2196 use_none: \prg_replicate:nn {#2} { n }
2197 \fi:
2198 \cs_end:
2199 }
2200 <latexrelease>\EndIncludeInRelease
2201 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_use:nnw}
2202 <latexrelease> {Hooks~with~args}

```

```

2203 <latexrelease>\cs_gset:Npn \hook_use:nnw #1 #2
2204 <latexrelease> { \use:c { use_none: \prg_replicate:nn {#2} { n } } }
2205 <latexrelease>\EndIncludeInRelease

```

(End of definition for \hook_use:nnw and _hook_use_initialized:nnw. This function is documented on page 15.)

_hook_post_initialization_defs:

```

2206 <latexrelease>\IncludeInRelease{2023/06/01}{\_hook_post_initialization_defs:}
2207 <latexrelease> {Hooks~with~args}
2208 \cs_new_protected:Npn \_hook_post_initialization_defs:
2209 {
2210   \cs_gset_eq:NN \hook_use:n \_hook_use_initialized:n
2211   \cs_gset_eq:NN \hook_use:nnw \_hook_use_initialized:nnw
2212   \cs_gset_eq:NN \_hook_preamble_hook:n \use_none:n
2213   \cs_gset_eq:NN \_hook_post_initialization_defs: \prg_do_nothing:
2214 }
2215 <latexrelease>\EndIncludeInRelease
2216 <latexrelease>\IncludeInRelease{2020/10/01}{\_hook_post_initialization_defs:}
2217 <latexrelease> {Hooks~with~args}
2218 <latexrelease>\cs_undefine:N \_hook_post_initialization_defs:
2219 <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 _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.

```

2220 <latexrelease>\IncludeInRelease{2021/11/15}{\_hook_use:wn}
2221 <latexrelease> {Standardise~generic~hook~names}
2222 <latexrelease>\EndIncludeInRelease
2223 <latexrelease>\IncludeInRelease{2020/10/01}{\_hook_use:wn}
2224 <latexrelease> {Standardise~generic~hook~names}
2225 <latexrelease>\cs_new:Npn \_hook_use:wn #1 / #2 \s_hook_mark #3
2226 <latexrelease> {
2227   \str_if_eq:nnTF {#1} { file }
2228   { \_hook_try_file_hook:n {#3} }
2229   { } % Hook doesn't exist
2230 }
2231 <latexrelease>\cs_new_protected:Npn \_hook_try_file_hook:n #1
2232 <latexrelease> {
2233   \_hook_if_file_hook:wTF #1 / \s_hook_mark
2234   {
2235     \exp_args:Ne \_hook_if_usable_use:n
2236     { \exp_args:Ne \_hook_file_hook_normalize:n {#1} }
2237   }
2238   { \_hook_if_usable_use:n {#1} } % file/ generic hook (e.g. file/before)
2239 }

```

```

2240 <latexrelease>\cs_new_protected:Npn \__hook_if_usable_use:n #1
2241 <latexrelease> {
2242 <latexrelease> \tl_if_exist:cT { __hook~#1 }
2243 <latexrelease> {
2244 <latexrelease> \__hook_preamble_hook:n {#1}
2245 <latexrelease> \cs:w __hook~#1 \cs_end:
2246 <latexrelease> }
2247 <latexrelease> }
2248 <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.

```

2249 <latexrelease>\IncludeInRelease{2023/06/01}{\hook_use_once:nnw}
2250 <latexrelease> {Hooks~with~args}
2251 \cs_new_protected:Npn \hook_use_once:n #1
2252 {
2253 \__hook_if_execute_immediately:nF {#1}
2254 { \__hook_normalize_hook_args:Nn \__hook_use_once:nn { \use:n {#1} } { 0 } }
2255 }
2256 \cs_new_protected:Npn \hook_use_once:nnw #1 #2
2257 {
2258 \__hook_if_execute_immediately:nF {#1}
2259 { \__hook_normalize_hook_args:Nn \__hook_use_once:nn { \use:n {#1} } {#2} }
2260 }
2261 <latexrelease>\EndIncludeInRelease

```

(End of definition for \hook_use_once:n and \hook_use_once:nnw. These functions are documented on page 15.)

```

2262 <latexrelease>\IncludeInRelease{2020/10/01}{\hook_use_once:nnw}
2263 <latexrelease> {Hooks~with~args}
2264 <latexrelease>\cs_gset_protected:Npn \hook_use_once:n #1
2265 <latexrelease> {
2266 <latexrelease> \__hook_if_execute_immediately:nF {#1}
2267 <latexrelease> { \__hook_normalize_hook_args:Nn \__hook_use_once:n { \use:n {#1} } }
2268 <latexrelease> }
2269 <latexrelease>\cs_gset:Npn \hook_use_once:nnw #1 #2
2270 <latexrelease> { \use:c { use_none: \prg_replicate:nn {#2} { n } } }
2271 <latexrelease>\EndIncludeInRelease

```

__hook_use_once:nn

```

2272 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_use_once:nn}
2273 <latexrelease> {Hooks~with~args}
2274 \cs_new_protected:Npn \__hook_use_once:nn #1 #2
2275 {
2276 \__hook_preamble_hook:n {#1}
2277 \__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.

```

2278 \__hook_replacing_args_false:
2279 \__hook_cs_gput_right:nnn { _next } {#1} { \__hook_use_once_clear:n {#1} }
2280 \__hook_replacing_args_reset:
2281 \__hook_if_usable:nTF {#1}
2282 { \__hook_use_initialized:n {#1} }
2283 {
2284 \int_compare:nNnT {#2} > { 0 }
2285 { \use:c { use_none: \prg_replicate:nn {#2} { n } } } }
2286 }
2287 }
2288 <latexrelease>\EndIncludeInRelease
2289 %
2290 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_use_once:nn}
2291 <latexrelease> {Hooks~with~args}
2292 <latexrelease>\cs_gset_protected:Npn \__hook_use_once:n #1
2293 <latexrelease> {
2294 <latexrelease> \__hook_preamble_hook:n {#1}
2295 <latexrelease> \__hook_use_once_set:n {#1}
2296 <latexrelease> \__hook_use_initialized:n {#1}
2297 <latexrelease> \__hook_use_once_clear:n {#1}
2298 <latexrelease> }
2299 <latexrelease>\cs_undefine:N \__hook_use_once:nn
2300 <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.

```

2301 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_use_once_clear:n}
2302 <latexrelease> {Hooks~with~args}
2303 \cs_new_protected:Npn \__hook_use_once_set:n #1
2304 { \__hook_tl_gset:cn { g__hook_#1_reversed_tl } { I } }
2305 \cs_new_protected:Npn \__hook_use_once_clear:n #1
2306 {
2307 \__hook_code_gset:nn {#1} { }
2308 \__hook_next_gset:nn {#1} { }
2309 \__hook_toplevel_gset:nn {#1} { }
2310 \prop_gclear_new:c { g__hook_#1_code_prop }
2311 }
2312 <latexrelease>\EndIncludeInRelease
2313 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_use_once_clear:n}
2314 <latexrelease> {Hooks~with~args}
2315 <latexrelease>\cs_new_protected:Npn \__hook_use_once_clear:n #1
2316 <latexrelease> {
2317 <latexrelease> \__hook_tl_gclear:c { __hook~#1 }
2318 <latexrelease> \__hook_tl_gclear:c { __hook_next~#1 }

```

```

2319 <latexrelease> \_\hook_t1_gclear:c { \_\hook_toplevel~#1 }
2320 <latexrelease> \prop_gclear_new:c { g_\_\hook_#1_code_prop }
2321 <latexrelease> }
2322 <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.

```

2323 \prg_new_conditional:Npnn \_\hook_if_execute_immediately:n #1 { T, F, TF }
2324 {
2325   \exp_after:wN \_\hook_use_none_delimit_by_s_mark:w
2326   \if:w I
2327     \if_cs_exist:w g_\_\hook_#1_reversed_tl \cs_end:
2328     \cs:w g_\_\hook_#1_reversed_tl \exp_after:wN \cs_end:
2329     \fi:
2330     X
2331     \s_\hook_mark \prg_return_true:
2332   \else:
2333     \s_\hook_mark \prg_return_false:
2334   \fi:
2335 }

```

(End of definition for _\hook_if_execute_immediately:nTF.)

4.10 Querying a hook

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

Hooks are a bit more complicated: they have several other states as discussed in 4.4.2. A hook may exist or not, and either way it may or may not be empty (even a hook that doesn’t exist may be non-empty) or may be disabled.

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

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

\hook_if_empty_p:n
\hook_if_empty:nTF

Test if a hook is empty (that is, no code was added to that hook). A `<hook>` being empty means that all three of its `\g__\hook_<hook>_code_prop`, its `_\hook_toplevel__\hook_<hook>` and its `_\hook_next__\hook_<hook>` are empty.

```

2336 <latexrelease> \IncludeInRelease{2023/06/01}{\hook_if_empty:n}
2337 <latexrelease> {Hooks~with~args}
2338 \prg_new_conditional:Npnn \hook_if_empty:n #1 { p , T , F , TF }
2339 {
2340   \if:w
2341     T
2342     \prop_if_exist:cT { g_\_\hook_#1_code_prop }
2343     { \prop_if_empty:cF { g_\_\hook_#1_code_prop } { F } }

```

```

2344     \__hook_cs_if_empty:cF { __hook_toplevel~#1 } { F }
2345     \__hook_cs_if_empty:cF { __hook_next~#1 } { F }
2346     T
2347     \prg_return_true:
2348   \else:
2349     \prg_return_false:
2350   \fi:
2351 }
2352 \<latexrelease>\EndIncludeInRelease

2353 \<latexrelease>\IncludeInRelease{2020/10/01}{\hook_if_empty:n}
2354 \<latexrelease>           {Hooks~with~args}
2355 \<latexrelease>\prg_new_conditional:Npnn \hook_if_empty:n #1 { p , T , F , TF }
2356 \<latexrelease> {
2357   \<latexrelease>   \__hook_if_structure_exist:nTF {#1}
2358   \<latexrelease>   {
2359     \<latexrelease>     \bool_lazy_and:nnTF
2360     \<latexrelease>     { \prop_if_empty_p:c { g__hook_#1_code_prop } }
2361     \<latexrelease>     {
2362       \<latexrelease>       \bool_lazy_and_p:nn
2363       \<latexrelease>       { \tl_if_empty_p:c { __hook_toplevel~#1 } }
2364       \<latexrelease>       { \tl_if_empty_p:c { __hook_next~#1 } }
2365     }
2366     \<latexrelease>     { \prg_return_true: }
2367     \<latexrelease>     { \prg_return_false: }
2368   }
2369   \<latexrelease>   { \prg_return_true: }
2370 }
2371 \<latexrelease>\EndIncludeInRelease

```

(End of definition for \hook_if_empty:nTF. This function is documented on page 16.)

__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).

```

2372 \prg_new_conditional:Npnn \__hook_if_usable:n #1 { p , T , F , TF }
2373 {
2374   \cs_if_exist:cTF { __hook~#1 }
2375   { \prg_return_true: }
2376   { \prg_return_false: }
2377 }

```

(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.

```

2378 \prg_new_conditional:Npnn \__hook_if_structure_exist:n #1 { p , T , F , TF }
2379 {
2380   \prop_if_exist:cTF { g__hook_#1_code_prop }
2381   { \prg_return_true: }
2382   { \prg_return_false: }
2383 }

```

(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 2384 \prg_new_conditional:Npnn \__hook_if_declared:n #1 { p, T, F, TF }
                        2385 {
                        2386   \tl_if_exist:cTF { g__hook_#1_declared_tl }
                        2387   { \prg_return_true: }
                        2388   { \prg_return_false: }
                        2389 }

```

(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 2390 \prg_new_conditional:Npnn \__hook_if_reversed:n #1 { p, T, F, TF }
                        2391 {
                        2392   \exp_after:wN \__hook_use_none_delimit_by_s_mark:w
                        2393   \if:w - \cs:w g__hook_#1_reversed_tl \cs_end:
                        2394   \s__hook_mark \prg_return_true:
                        2395   \else:
                        2396   \s__hook_mark \prg_return_false:
                        2397   \fi:
                        2398 }

```

(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 2399 \prg_new_conditional:Npnn \__hook_if_generic:n #1 { T, TF }
\hook_if_deprecated_generic_p:n 2400 { \__hook_if_generic:w #1 / / / \s__hook_mark }
\hook_if_deprecated_generic:nTF 2401 \cs_new:Npn \__hook_if_generic:w #1 / #2 / #3 / #4 \s__hook_mark
                                2402 {
                                2403   \cs_if_exist:cTF { c__hook_generic_#1/./#3_tl }
                                2404   { \prg_return_true: }
                                2405   { \prg_return_false: }
                                2406 }
                                2407 \prg_new_conditional:Npnn \__hook_if_deprecated_generic:n #1 { T, TF }
                                2408 { \__hook_if_deprecated_generic:w #1 / / / \s__hook_mark }
                                2409 \cs_new:Npn \__hook_if_deprecated_generic:w #1 / #2 / #3 / #4 \s__hook_mark
                                2410 {
                                2411   \cs_if_exist:cTF { c__hook_deprecated_#1/./#2_tl }
                                2412   {
                                2413     \tl_if_empty:nTF {#3}
                                2414     { \prg_return_false: }
                                2415     { \prg_return_true: }
                                2416   }
                                2417   { \prg_return_false: }
                                2418 }

```

(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 2419 <latexrelease>\IncludeInRelease{2023/06/01}{\__hook_if_cmd_hook:n}
\__hook_if_cmd_hook_p:w 2420 <latexrelease>{Hooks~with~args}
\__hook_if_cmd_hook:wTF 2421 \prg_new_conditional:Npnn \__hook_if_cmd_hook:n #1 { T }

```



```

2422 { \__hook_if_cmd_hook:w #1 / / / \s__hook_mark }
2423 \cs_new:Npn \__hook_if_cmd_hook:w #1 / #2 / #3 / #4 \s__hook_mark
2424 {
2425   \if:w Y
2426     \str_if_eq:nnF {#1} { cmd } { N }
2427     \tl_if_exist:cF { c__hook_generic_#1/./#3_tl } { N }
2428     Y
2429     \prg_return_true:
2430   \else:
2431     \prg_return_false:
2432   \fi:
2433 }
2434 <latexrelease>\EndIncludeInRelease
2435 <latexrelease>\IncludeInRelease{2020/10/01}{\__hook_if_cmd_hook:n}
2436 <latexrelease>{Hooks~with~args}
2437 <latexrelease>\cs_undefine:N \__hook_if_cmd_hook:nT
2438 <latexrelease>\EndIncludeInRelease

```

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

__hook_if_generic_reversed_p:n
 __hook_if_generic_reversed:nTF

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

```

2439 \prg_new_conditional:Nppn \__hook_if_generic_reversed:n #1 { T }
2440 { \__hook_if_generic_reversed:w #1 / / / \scan_stop: }
2441 \cs_new:Npn \__hook_if_generic_reversed:w #1 / #2 / #3 / #4 \scan_stop:
2442 {
2443   \if_charcode:w - \cs:w c__hook_generic_#1/./#3_tl \cs_end:
2444   \prg_return_true:
2445   \else:
2446   \prg_return_false:
2447   \fi:
2448 }

```

(End of definition for __hook_if_generic_reversed:nTF.)

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

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

```

2449 \seq_new:N \g__hook_replacing_stack_seq
2450 \cs_new:Npn \__hook_misused_if_replacing_args:nn #1 #2
2451 {
2452   \msg_expandable_error:nnn { latex2e } { should-not-happen }
2453   { Misused~\__hook_if_replacing_args:. }
2454 }
2455 \cs_new:Npn \__hook_if_replacing_args:TF
2456 { \__hook_misused_if_replacing_args:nn }
2457 \cs_new_protected:Npn \__hook_replacing_args_true:
2458 {
2459   \seq_gpush:No \g__hook_replacing_stack_seq
2460   { \__hook_if_replacing_args:TF }
2461   \cs_set:Npn \__hook_if_replacing_args:TF { \use_i:nn }
2462 }
2463 \cs_new_protected:Npn \__hook_replacing_args_false:
2464 {
2465   \seq_gpush:No \g__hook_replacing_stack_seq
2466   { \__hook_if_replacing_args:TF }

```

```

2467 \cs_set:Npn \__hook_if_replacing_args:TF { \use_ii:nn }
2468 }
2469 \cs_new_protected:Npn \__hook_replacing_args_reset:
2470 {
2471   \seq_gpop:NN \g__hook_replacing_stack_seq \l__hook_return_tl
2472   \cs_gset_eq:NN \__hook_if_replacing_args:TF \l__hook_return_tl
2473 }

```

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

4.11 Messages

Hook errors are LaTeX kernel errors:

```

2474 \prop_gput:Nnn \g_msg_module_type_prop { hooks } { LaTeX }

```

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

```

2475 \prop_gput:Nnn \g_msg_module_type_prop { latex2e } { LaTeX }
2476 \prop_gput:Nnn \g_msg_module_name_prop { latex2e } { kernel }

```

```

2477 \msg_new:nnnn { hooks } { labels-incompatible }
2478 {
2479   Labels~'#1'~and~'#2'~are~incompatible
2480   \str_if_eq:nnF {#3} {??} { ~in-hook~'#3' } .~
2481   \int_compare:nNnTF {#4} = { 1 }
2482     { The~ code~ for~ both~ labels~ will~ be~ dropped. }
2483     { You~ may~ see~ errors~ later. }
2484 }
2485 { LaTeX-found-two-incompatible-labels-in-the-same-hook.~
2486   This~indicates~an~incompatibility~between~packages. }

2487 \msg_new:nnnn { hooks } { exists }
2488 { Hook~'#1'~ has~ already~ been~ declared. }
2489 { There~ already~ exists~ a~ hook~ declaration~ with~ this~
2490   name.\\
2491   Please~ use~ a~ different~ name~ for~ your~ hook.}

2492 <latexrelease>\IncludeInRelease{2023/06/01}{too-many-args}
2493 <latexrelease>{Hooks~with~args}

```

```

2494 \msg_new:nnnn { hooks } { too-many-args }
2495 { Too~many~arguments~for~hook~'#1'. }
2496 {
2497   You~tried~to~declare~a~hook~with~#2~arguments,~but~a~
2498   hook~can~only~have~up~to~nine.~LaTeX~will~define~this~
2499   hook~with~nine~arguments.
2500 }

2501 \msg_new:nnnn { hooks } { without-args }
2502 { Hook~'#1'~has~no~arguments. }
2503 {
2504   You~tried~to~use~\iow_char:N\\#2WithArguments~
2505   on~a~hook~that~takes~no~arguments.\\
2506   Check~the~usage~of~the~hook~or~use~\iow_char:N\\#2~instead.\\
2507   \\
2508   LaTeX~will~use~\iow_char:N\\#2.
2509 }

```

```

2510 \msg_new:nnnn { hooks } { one-time-args }
2511 { You~can't~have~arguments~in~used~one~time~hook~'#1'. }
2512 {
2513   You~tried~to~use~\iow_char:N\\#2WithArguments~
2514   on~a~one~time~hook~that~has~already~been~used.~
2515   You~have~to~add~the~code~before~the~hook~is~used,~
2516   or~add~the~code~without~arguments~using~\iow_char:N\\#2~instead.\\
2517   \\
2518   LaTeX~will~use~\iow_char:N\\#2.
2519 }

2520 \latexrelease\EndIncludeInRelease
2521 \latexrelease\IncludeInRelease{2020/10/01}{too-many-args}
2522 \latexrelease{Hooks~with~args}
2523 \latexrelease\EndIncludeInRelease

2524 \msg_new:nnnn { hooks } { hook-disabled }
2525 { Cannot~add~code~to~disabled~hook~'#1'. }
2526 {
2527   The~hook~'#1'~you~tried~to~add~code~to~was~previously~disabled~
2528   with~\iow_char:N\\hook_disable_generic:n~or~\iow_char:N\\DisableGenericHook,~so~
2529   it~cannot~have~code~added~to~it.
2530 }

2531 \msg_new:nnn { hooks } { empty-label }
2532 {
2533   Empty~code~label~\msg_line_context:~
2534   Using~'\_hook_currname_or_default:'~instead.
2535 }

2536 \msg_new:nnn { hooks } { no-default-label }
2537 {
2538   Missing~(empty)~default~label~\msg_line_context:. \\
2539   This~command~was~ignored.
2540 }

2541 \msg_new:nnnn { hooks } { unknown-rule }
2542 {
2543   Unknown~ relationship~ '#3'~
2544   between~ labels~ '#2'~ and~ '#4'~
2545   \str_if_eq:nnF {#1} {??} { ~in~hook~'#1' }. ~
2546   Perhaps~ a~ misspelling?
2547 }
2548 {
2549   The~ relation~ used~ not~ known~ to~ the~ system.~ Allowed~ values~ are~
2550   'before'~ or~ '<',~
2551   'after'~ or~ '>',~
2552   'incompatible-warning',~
2553   'incompatible-error',~
2554   'voids'~ or~
2555   'unrelated'.
2556 }

2557 \msg_new:nnnn { hooks } { rule-too-late }
2558 {
2559   Sorting~rule~for~'#1'~hook~applied~too~late.\\
2560   Try~setting~this~rule~earlier.
2561 }

```

```

2562 {
2563   You~tried~to~set~the~ordering~of~hook~'#1'~using\\
2564   \ \ \iow_char:N\\DeclareHookRule{#1}{#2}{#3}{#4}\\
2565   but~hook~'#1'~was~already~used~as~a~one-time~hook,~
2566   thus~sorting~is\\
2567   no~longer~possible.~Declare~the~rule~
2568   before~the~hook~is~used.
2569 }

2570 \msg_new:nnnn { hooks } { misused-top-level }
2571 {
2572   Illegal~use~of~\iow_char:N \\AddToHook{#1}[top-level]{...}.\\
2573   'top-level'~is~reserved~for~the~user's~document.
2574 }
2575 {
2576   The~'top-level'~label~is~meant~for~user~code~only,~and~should~only~
2577   be~used~(sparingly)~in~the~main~document.~Use~the~default~label~
2578   '\_hook_currname_or_default:'~for~this~\@cls@pkg,~or~another~
2579   suitable~label.
2580 }

2581 \msg_new:nnn { hooks } { set-top-level }
2582 {
2583   You~cannot~change~the~default~label~'#1'~'top-level'.~Illegal \\
2584   \use:nn { ~ } { ~ } \iow_char:N \\#2{#3} \\
2585   \msg_line_context:.
2586 }

2587 \msg_new:nnn { hooks } { extra-pop-label }
2588 {
2589   Extra~\iow_char:N \\PopDefaultHookLabel. \\
2590   This~command~will~be~ignored.
2591 }
2592 \msg_new:nnn { hooks } { missing-pop-label }
2593 {
2594   Missing~\iow_char:N \\PopDefaultHookLabel. \\
2595   The~label~'#1'~was~pushed~but~never~popped.~Something~is~wrong.
2596 }

2597 \msg_new:nnn { latex2e } { should-not-happen }
2598 {
2599   This~should~not~happen.~#1 \\
2600   Please~report~at~https://github.com/latex3/latex2e.
2601 }

2602 \msg_new:nnn { hooks } { activate-disabled }
2603 {
2604   Cannot~ activate~ hook~ '#1'~ because~ it~ is~ disabled!
2605 }

2606 \msg_new:nnn { hooks } { cannot-remove }
2607 {
2608   Cannot~remove~chunk~'#2'~from~hook~'#1'~because~
2609   \_hook_if_structure_exist:nTF {#1}
2610   { it~does~not~exist~in~that~hook. }
2611   { the~hook~does~not~exist. }
2612 }

```

```

2613 \msg_new:nnn { hooks } { generic-deprecated }
2614 {
2615   Generic~hook~'~#1/#2/#3'~is~deprecated. \\
2616   Use~hook~'~#1/#3/#2'~instead.
2617 }

```

4.12 L^AT_EX 2_ε package interface commands

\NewHook Declaring new hooks ...

```

\NewReversedHook 2618 \NewDocumentCommand \NewHook { m }
\NewMirroredHookPair 2619 { \hook_new:n {#1} }
2620 \NewDocumentCommand \NewReversedHook { m }
2621 { \hook_new_reversed:n {#1} }
2622 \NewDocumentCommand \NewMirroredHookPair { mm }
2623 { \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 2624 <latexrelease>\IncludeInRelease{2023/06/01}{\NewHookWithArguments}
\NewMirroredHookPairWithArguments 2625 <latexrelease> {Hooks~with~args}
2626 \NewDocumentCommand \NewHookWithArguments { mm }
2627 { \hook_new_with_args:nn {#1} {#2} }
2628 \NewDocumentCommand \NewReversedHookWithArguments { mm }
2629 { \hook_new_reversed_with_args:nn {#1} {#2} }
2630 \NewDocumentCommand \NewMirroredHookPairWithArguments { mmm }
2631 { \hook_new_pair_with_args:nnn {#1} {#2} {#3} }
2632 <latexrelease>\EndIncludeInRelease
2633 <latexrelease>\IncludeInRelease{2020/10/01}{\NewHookWithArguments}
2634 <latexrelease> {Hooks~with~args}
2635 <latexrelease>\cs_new_protected:Npn \NewHookWithArguments #1 #2 { }
2636 <latexrelease>\cs_new_protected:Npn \NewReversedHookWithArguments #1 #2 { }
2637 <latexrelease>\cs_new_protected:Npn \NewMirroredHookPairWithArguments #1 #2 #3 { }
2638 <latexrelease>\EndIncludeInRelease

```

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

```

2639 <latexrelease>\IncludeInRelease{2021/06/01}{\ActivateGenericHook}
2640 <latexrelease> {Providing~hooks}

```

\ActivateGenericHook Providing new hooks ...

```

2641 \NewDocumentCommand \ActivateGenericHook { m }
2642 { \hook_activate_generic:n {#1} }

```

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

\DisableGenericHook Disabling a generic hook.

```

2643 \NewDocumentCommand \DisableGenericHook { m }
2644 { \hook_disable_generic:n {#1} }

```

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

```

2645 <latexrelease>\EndIncludeInRelease

```

```

2646 <latexrelease>\IncludeInRelease{2020/10/01}{\ActivateGenericHook}
2647 <latexrelease>{Providing-hooks}
2648 <latexrelease>\def \ActivateGenericHook #1 { }
2649 <latexrelease>\def \DisableGenericHook #1 { }
2650 <latexrelease>\EndIncludeInRelease

```

\AddToHook

\AddToHookWithArguments

```

2651 <latexrelease>\IncludeInRelease{2023/06/01}{\AddToHookWithArguments}
2652 <latexrelease>{Hooks-with-args}
2653 \NewDocumentCommand \AddToHook { m o +m }
2654 { \hook_gput_code:nnn {#1} {#2} {#3} }
2655 \NewDocumentCommand \AddToHookWithArguments { m o +m }
2656 { \hook_gput_code_with_args:nnn {#1} {#2} {#3} }
2657 <latexrelease>\EndIncludeInRelease
2658 <latexrelease>\IncludeInRelease{2020/10/01}{\AddToHookWithArguments}
2659 <latexrelease>{Hooks-with-args}
2660 <latexrelease>\cs_new_protected:Npn \AddToHookWithArguments #1 #2 #3 { }
2661 <latexrelease>\EndIncludeInRelease

```

(End of definition for \AddToHook and \AddToHookWithArguments. These functions are documented on page 5.)

\AddToHookNext

\AddToHookNextWithArguments

```

2662 <latexrelease>\IncludeInRelease{2023/06/01}{\AddToHookNextWithArguments}
2663 <latexrelease>{Hooks-with-args}
2664 \NewDocumentCommand \AddToHookNext { m +m }
2665 { \hook_gput_next_code:nn {#1} {#2} }
2666 \NewDocumentCommand \AddToHookNextWithArguments { m +m }
2667 { \hook_gput_next_code_with_args:nn {#1} {#2} }
2668 <latexrelease>\EndIncludeInRelease
2669 <latexrelease>\IncludeInRelease{2020/10/01}{\AddToHookNextWithArguments}
2670 <latexrelease>{Hooks-with-args}
2671 <latexrelease>\cs_new_protected:Npn \AddToHookNextWithArguments #1 #2 { }
2672 <latexrelease>\EndIncludeInRelease

```

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

\ClearHookNext

```

2673 \NewDocumentCommand \ClearHookNext { m }
2674 { \hook_gclear_next_code:n {#1} }

```

(End of definition for \ClearHookNext. This function is documented on page 7.)

\RemoveFromHook

```

2675 \NewDocumentCommand \RemoveFromHook { m o }
2676 { \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.

```

2677 \NewDocumentCommand \SetDefaultHookLabel { m }
2678 { \__hook_set_default_hook_label:n {#1} }
2679 %

```

```

2680 % The label is only automatically updated with \cs{@onefilewithoptions}
2681 % (\cs{usepackage} and \cs{documentclass}), but some packages, like
2682 % Ti\emph{k}Z, define package-like interfaces, like
2683 % \cs{usetikzlibrary} that are wrappers around \cs{input}, so they
2684 % inherit the default label currently in force (usually |top-level|,
2685 % but it may change if loaded in another package). To provide a
2686 % package-like behavior also for hooks in these files, we provide
2687 % high-level access to the default label stack.
2688 % \begin{macrocode}
2689 \NewDocumentCommand \PushDefaultHookLabel { m }
2690 { \__hook_curr_name_push:n {#1} }
2691 \NewDocumentCommand \PopDefaultHookLabel { }
2692 { \__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.

```

2693 \cs_gset_protected:Npn \@expl@push@filename@aux@@ #1#2#3
2694 {
2695   \__hook_curr_name_push:n {#3}
2696   \str_gset:Nx \g_file_curr_name_str {#3}
2697   #1 #2 {#3}
2698 }

```

(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

```

2699 <latexrelease> \IncludeInRelease{2023/06/01}{\UseHookWithArguments}
2700 <latexrelease> {Hooks~with~args}
2701 \cs_new:Npn \UseHook { \hook_use:n }
2702 \cs_new:Npn \UseOneTimeHook { \hook_use_once:n }
2703 \cs_new:Npn \UseHookWithArguments { \hook_use:nw }
2704 \cs_new:Npn \UseOneTimeHookWithArguments { \hook_use_once:nw }
2705 <latexrelease> \EndIncludeInRelease
2706 <latexrelease> \IncludeInRelease{2020/10/01}{\UseHookWithArguments}
2707 <latexrelease> {Hooks~with~args}
2708 <latexrelease> \cs_new:Npn \UseHookWithArguments #1 #2 { }
2709 <latexrelease> \cs_new:Npn \UseOneTimeHookWithArguments #1 #2 { }
2710 <latexrelease> \EndIncludeInRelease

```

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

\ShowHook

\LogHook

```

2711 \cs_new_protected:Npn \ShowHook { \hook_show:n }
2712 \cs_new_protected:Npn \LogHook { \hook_log:n }

```

(End of definition for \ShowHook and \LogHook. These functions are documented on page 12.)

\DebugHooksOn

\DebugHooksOff

```

2713 \cs_new_protected:Npn \DebugHooksOn { \hook_debug_on: }
2714 \cs_new_protected:Npn \DebugHooksOff { \hook_debug_off: }

```

(End of definition for `\DebugHooksOn` and `\DebugHooksOff`. These functions are documented on page 13.)

`\DeclareHookRule`

```
2715 \NewDocumentCommand \DeclareHookRule { m m m m }
2716 { \hook_gset_rule:nnnn {#1}{#2}{#3}{#4} }
```

(End of definition for `\DeclareHookRule`. This function is documented on page 11.)

`\DeclareDefaultHookRule`

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

```
2717 \NewDocumentCommand \DeclareDefaultHookRule { m m m }
2718 { \hook_gset_rule:nnnn {??}{#1}{#2}{#3} }
2719 \@onlypreamble\DeclareDefaultHookRule
```

(End of definition for `\DeclareDefaultHookRule`. This function is documented on page 11.)

`\ClearHookRule`

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

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

```
2720 \NewDocumentCommand \ClearHookRule { m m m }
2721 { \hook_gset_rule:nnnn {#1}{#2}{unrelated}{#3} }
```

(End of definition for `\ClearHookRule`. This function is documented on page 11.)

`\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.

```
2722 \cs_new_eq:NN \IfHookEmptyTF \hook_if_empty:nTF
```

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

`\IfHookExistsTF`

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

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

```
2723 \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`

```
2724 \cs_new_protected:Npn \hook_disable:n
```

`\hook_provide_reversed:n`

```
2725 {
```

`\hook_provide_pair:nn`

```
2726 { \__hook_deprecated_warn:nn
```

`__hook_activate_generic_reversed:n`

```
2727 { hook_disable:n }
```

`__hook_activate_generic_pair:nn`

```
2728 { hook_disable_generic:n }
```

```
2729 \hook_disable_generic:n
```

```
2730 }
```

```
2731 \cs_new_protected:Npn \hook_provide:n
```

```
2732 {
```

```
2733 { \__hook_deprecated_warn:nn
```

```
2734 { hook_provide:n }
```



```

2735     { hook_activate_generic:n }
2736     \hook_activate_generic:n
2737   }
2738   \cs_new_protected:Npn \hook_provide_reversed:n
2739   {
2740     \__hook_deprecated_warn:nn
2741     { hook_provide_reversed:n }
2742     { hook_activate_generic:n }
2743     \__hook_activate_generic_reversed:n
2744   }
2745   \cs_new_protected:Npn \hook_provide_pair:nn
2746   {
2747     \__hook_deprecated_warn:nn
2748     { hook_provide_pair:nn }
2749     { hook_activate_generic:n }
2750     \__hook_activate_generic_pair:nn
2751   }
2752   \cs_new_protected:Npn \__hook_activate_generic_reversed:n #1
2753   { \__hook_normalize_hook_args:Nn \__hook_activate_generic:nn {#1} { - } }
2754   \cs_new_protected:Npn \__hook_activate_generic_pair:nn #1#2
2755   { \hook_activate_generic:n {#1} \__hook_activate_generic_reversed:n {#2} }

```

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

```

\DisableHook      Deprecated.
\ProvideHook      2756 \cs_new_protected:Npn \DisableHook
\ProvideReversedHook 2757 {
\ProvideMirroredHookPair 2758   \__hook_deprecated_warn:nn
2759   { DisableHook }
2760   { DisableGenericHook }
2761   \hook_disable_generic:n
2762 }
2763 \cs_new_protected:Npn \ProvideHook
2764 {
2765   \__hook_deprecated_warn:nn
2766   { ProvideHook }
2767   { ActivateGenericHook }
2768   \hook_activate_generic:n
2769 }
2770 \cs_new_protected:Npn \ProvideReversedHook
2771 {
2772   \__hook_deprecated_warn:nn
2773   { ProvideReversedHook }
2774   { ActivateGenericHook }
2775   \__hook_activate_generic_reversed:n
2776 }
2777 \cs_new_protected:Npn \ProvideMirroredHookPair
2778 {
2779   \__hook_deprecated_warn:nn
2780   { ProvideMirroredHookPair }
2781   { ActivateGenericHook }
2782   \__hook_activate_generic_pair:nn
2783 }

```

(End of definition for \DisableHook and others.)

```

\__hook_deprecated_warn:nn Warns about a deprecation, telling what should be used instead.
2784 \cs_new_protected:Npn \__hook_deprecated_warn:nn #1 #2
2785 { \msg_warning:nnnn { hooks } { deprecated } {#1} {#2} }
2786 \msg_new:nnn { hooks } { deprecated }
2787 {
2788   Command~\iow_char:N\|#1~is~deprecated~and~will~be~removed~in~a~
2789   future~release. \\ \\
2790   Use~\iow_char:N\|#2~instead.
2791 }

```

(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.

```

2792 \@@=
\@expl@@@initialize@all@@
\@expl@@@hook@curr@name@pop@@
2793 \cs_new_eq:NN \@expl@@@initialize@all@@
2794 \__hook_initialize_all:
2795 \cs_new_eq:NN \@expl@@@hook@curr@name@pop@@
2796 \__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.

```

2797 %
2798 \<latexrelease>\IncludeInRelease{0000/00/00}{lthooks}
2799 \<latexrelease>{The-hook-management}%
2800 \<latexrelease>
2801 \<latexrelease>\def \NewHook#1{}
2802 \<latexrelease>\def \NewReversedHook#1{}
2803 \<latexrelease>\def \NewMirroredHookPair#1#2{}
2804 \<latexrelease>
2805 \<latexrelease>\def \DisableGenericHook #1{}
2806 \<latexrelease>
2807 \<latexrelease>\long\def\AddToHookNext#1#2{}
2808 \<latexrelease>
2809 \<latexrelease>\def\AddToHook#1{\@gobble\AddToHook@args}
2810 \<latexrelease>\providecommand\@gobble\AddToHook@args[2] [] {}
2811 \<latexrelease>
2812 \<latexrelease>\def\RemoveFromHook#1{\@gobble\RemoveFromHook@arg}
2813 \<latexrelease>\providecommand\@gobble\RemoveFromHook@arg[1] [] {}
2814 \<latexrelease>
2815 \<latexrelease>\def \UseHook #1{}
2816 \<latexrelease>\def \UseOneTimeHook #1{}
2817 \<latexrelease>\def \ShowHook #1{}
2818 \<latexrelease>\let \DebugHooksOn \@empty
2819 \<latexrelease>\let \DebugHooksOff\@empty
2820 \<latexrelease>

```

```

2821 <latexrelease>\def \DeclareHookRule #1#2#3#4{}
2822 <latexrelease>\def \DeclareDefaultHookRule #1#2#3{}
2823 <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.

```

2824 <latexrelease>\long\def \IfHookExistsTF #1#2#3{#3}
2825 <latexrelease>\long\def \IfHookEmptyTF #1#2#3{#2}
2826 <latexrelease>
2827 <latexrelease>\EndModuleRelease
2828 <@@=hook>

2829 <latexrelease>\cs:w __hook_rollback_tidying: \cs_end:
2830 <latexrelease>\bool_lazy_and:nnT
2831 <latexrelease>    { \int_compare_p:nNn { \sourceLaTeXdate } > { 20230600 } }
2832 <latexrelease>    { \int_compare_p:nNn { \requestedLaTeXdate } < { 20230601 } }
2833 <latexrelease> {
2834 <latexrelease>   \cs_gset_protected:Npn \__hook_rollback_tidying:
2835 <latexrelease>   {
2836 <latexrelease>     \@latex@error { Rollback-code-executed-twice }
2837 <latexrelease>     {
2838 <latexrelease>       Something-went-wrong~(unless-this-was~
2839 <latexrelease>       done-on-purpose-in-a-testing-environment).
2840 <latexrelease>     }
2841 <latexrelease>     \use_none:nnnn
2842 <latexrelease>   }
2843 <latexrelease>   \cs_set:Npn \__hook_tmp:w #1 #2
2844 <latexrelease>   {
2845 <latexrelease>     \__hook_tl_gset:cx { __hook#1~#2 }
2846 <latexrelease>     {
2847 <latexrelease>       \exp_args:No \exp_not:o
2848 <latexrelease>       {
2849 <latexrelease>         \cs:w __hook#1~#2 \exp_last_unbraced:Ne \cs_end:
2850 <latexrelease>         { \__hook_braced_cs_parameter:n { __hook#1~#2 } }
2851 <latexrelease>       }
2852 <latexrelease>     }
2853 <latexrelease>   }
2854 <latexrelease>   \seq_map_inline:Nn \g__hook_all_seq
2855 <latexrelease>   {
2856 <latexrelease>     \exp_after:wN \cs_gset_nopar:Npn
2857 <latexrelease>     \cs:w g__hook_#1_code_prop \exp_args:NNo \exp_args:No
2858 <latexrelease>     \cs_end: { \cs:w g__hook_#1_code_prop \cs_end: }
2859 <latexrelease>     \__hook_tmp:w { _toplevel } {#1}
2860 <latexrelease>     \__hook_tmp:w { _next } {#1}
2861 <latexrelease>   }
2862 <latexrelease> }
2863 <latexrelease> \ExplSyntaxOff
2864 </2ekernel | latexrelease>
2865 <@@=)

```

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>\@@\textvisiblespace_\meta_{hook}</code> ..	<i>32</i>
<code>\@@_next\textvisiblespace_\meta_{hook}</code>	<i>32</i>
<code>\@@_toplevel\textvisiblespace_\meta_{hook}</code>	<i>32</i>
<code>\</code>	1708, 1719, 2490, 2504, 2505, 2506, 2507, 2508, 2513, 2516, 2517, 2518, 2528, 2538, 2559, 2563, 2564, 2566, 2572, 2583, 2584, 2589, 2594, 2599, 2615, 2788, 2789, 2790
<code>\(addto-cmd)</code>	5
<code>_</code>	2564
A	
<code>\ActivateGenericHook</code>	2639, <u>2641</u> , 2646, 2648, <i>20</i>
<code>\AddToHook</code>	<u>2651</u> , 2809, <i>5</i>
<code>\AddToHookNext</code>	<u>2662</u> , 2807, <i>7</i>
<code>\AddToHookNextWithArguments</code>	<u>2662</u> , <i>7</i>
<code>\AddToHookWithArguments</code>	<u>2651</u> , <i>6</i>
<code>\AfterEndEnvironment</code>	25
<code>\AtBeginDocument</code>	19
<code>\AtBeginEnvironment</code>	25
<code>\AtEndDocument</code>	26
<code>\AtEndEnvironment</code>	25
<code>\AtEndPreamble</code>	26
B	
<code>\BeforeBeginEnvironment</code>	25
<code>\begin</code>	2688, <i>56</i>
<code>\bfdefault</code>	28
<code>\bfseries</code>	28
bool commands:	
<code>\bool_gset_false:N</code>	15
<code>\bool_gset_true:N</code>	10
<code>\bool_if:NTF</code> 1842, 1851, 1937, 1946, 21	
<code>\bool_lazy_and:nnTF</code>	1844, 1939, 2359, 2830, 275
<code>\bool_lazy_and_p:nn</code>	2362
<code>\bool_lazy_or:nnTF</code>	856
<code>\bool_new:N</code>	6, 24
<code>\bool_set_false:N</code>	1834, 1929
<code>\bool_set_true:N</code>	1831, 1926
<code>\bool_while_do:nn</code>	1533, 1614
<code>\c_false_bool</code>	50
C	
<code>\ClearHookNext</code>	<u>2673</u> , <i>7</i>
<code>\ClearHookRule</code>	<u>2720</u> , 2823, <i>11</i>
<code>\clearpage</code>	27
clist commands:	
<code>\clist_gclear:N</code>	1532, 1613
<code>\clist_gput_left:Nn</code>	1443, 1475
<code>\clist_gput_right:Nn</code>	1445, 1477
<code>\clist_if_empty:NTF</code>	1861, 1956
<code>\clist_if_exist:NTF</code>	121
<code>\clist_map_inline:nn</code>	887, 896
<code>\clist_new:N</code>	123, 140
<code>\clist_use:Nn</code>	1863, 1958
cs commands:	
<code>\cs:w</code>	1026, 1083, 1233, 1366, 1468, 1469, 1524, 1544, 1547, 1605, 1625, 1628, 1644, 1645, 1669, 1670, 1671, 1678, 1685, 2122, 2141, 2147, 2161, 2171, 2192, 2245, 2328, 2393, 2443, 2829, 2849, 2857, 2858, 379, <i>43</i>
<code>\cs_end:</code> ...	747, 1026, 1083, 1236, 1369, 1468, 1469, 1524, 1544, 1547, 1605, 1625, 1628, 1644, 1645, 1660, 1670, 1671, 1678, 1685, 2016, 2121, 2125, 2129, 2141, 2146, 2147, 2161, 2167, 2180, 2193, 2198, 2245, 2327, 2328, 2393, 2443, 2829, 2849, 2858, 379
<code>\cs_generate_variant:Nn</code>	1040, 1063, 1573, 1647, 37, 38, 39, 40, 41, 42, 43, 53, 54, 57, 66
<code>\cs_gset:Npn</code>	1039, 1059, 1569, 1980, 2203, 2269
<code>\cs_gset:Npx</code>	1008
<code>\cs_gset_eq:NN</code>	750, 1376, 1402, 1415, 1416, 1419, 2110, 2210, 2211, 2212, 2213, 2472
<code>\cs_gset_nopar:Npn</code>	2856
<code>\cs_gset_nopar:Npx</code>	50, 52
<code>\cs_gset_protected:Npn</code>	578, 580, 597, 621, 668, 675, 1400, 1457, 2056, 2058, 2086, 2091, 2109, 2264, 2292, 2693, 2834, 88, 91, 100, 133, 160, 188, 191, 197, 221, 226, 251, 310, 328
<code>\cs_gset_protected:Npx</code>	1120, 20
<code>\cs_if_exist:NTF</code>	1011, 1076, 2374, 2403, 2411
<code>\cs_if_exist_p:N</code>	277
<code>\cs_if_exist_use:NTF</code> 1242, 1286, 1311	

<code>\cs_new:Npn</code>	874,		
	876, 878, 1161, 1173, 1178, 1186,		
	1195, 1197, 1204, 1231, 1238, 1240,		
	1348, 1364, 1487, 1488, 1649, 1650,		
	1976, 1983, 2119, 2144, 2152, 2165,		
	2173, 2181, 2190, 2225, 2401, 2409,		
	2423, 2441, 2450, 2455, 2701, 2702,		
	2703, 2704, 2708, 2709, 45, 46,		
	330, 336, 349, 359, 360, 362, 376, 382		
<code>\cs_new_eq:NN</code>			
	. . . 1145, 1327, 1333, 1371, 1724,		
	1725, 1726, 1727, 1728, 1729, 2180,		
	2722, 2723, 2793, 2795, 7, 23, 34, 63		
<code>\cs_new_protected:Npn</code> . .	485, 491,		
	497, 513, 526, 552, 569, 625, 644,		
	651, 658, 685, 690, 695, 704, 742,		
	744, 752, 754, 757, 759, 927, 929,		
	962, 995, 1007, 1009, 1038, 1051,		
	1053, 1055, 1057, 1074, 1098, 1105,		
	1262, 1269, 1300, 1322, 1328, 1334,		
	1339, 1342, 1345, 1346, 1374, 1425,		
	1496, 1577, 1651, 1658, 1667, 1674,		
	1681, 1688, 1696, 1702, 1713, 1730,		
	1737, 1749, 1754, 1759, 1761, 1765,		
	1881, 1994, 2009, 2014, 2023, 2041,		
	2047, 2060, 2073, 2100, 2104, 2114,		
	2127, 2136, 2150, 2156, 2178, 2185,		
	2208, 2231, 2240, 2251, 2256, 2274,		
	2303, 2305, 2315, 2457, 2463, 2469,		
	2635, 2636, 2637, 2660, 2671, 2711,		
	2712, 2713, 2714, 2724, 2731, 2738,		
	2745, 2752, 2754, 2756, 2763, 2770,		
	2777, 2784, 8, 13, 18, 47, 49, 51,		
	55, 58, 64, 69, 71, 73, 104, 148,		
	172, 174, 176, 201, 203, 205, 231,		
	266, 268, 285, 290, 292, 383, 392,		
	397, 405, 429, 431, 447, 455, 465, 474		
<code>\cs_parameter_spec:N</code>	1189		
<code>\cs_replacement_spec</code>	59		
<code>\cs_replacement_spec:N</code>	1164,		
	1392, 1412, 1805, 1818, 1827, 37, 41		
<code>\cs_set:Npn</code>	1107, 1118,		
	1124, 1133, 1802, 2461, 2467, 2843		
<code>\cs_set_eq:NN</code> 1442, 1443, 1444, 1445,			
	1474, 1475, 1476, 1477, 1751, 1756		
<code>\cs_set_nopar:Npx</code>	48		
<code>\cs_set_protected:Npn</code> 1996, 2026, 418			
<code>\cs_to_str:N</code>	43		
<code>\cs_undefine:N</code>	642,		
	1044, 1045, 1046, 1047, 1067, 1068,		
	1069, 1070, 1094, 1149, 1169, 1227,		
	1248, 1249, 1260, 1347, 2218, 2299,		
	2437, 79, 80, 90, 132, 190, 196, 271		
<code>\csname</code>	8		
		D	
debug commands:			
	<code>\debug_resume:</code>	31	
	<code>\debug_suspend:</code>	31	
<code>\DebugHooksOff</code>	2713, 2819, 13		
<code>\DebugHooksOn</code>	2713, 2818, 13		
<code>\DeclareDefaultHookRule</code> .	2717, 2822, 11		
<code>\DeclareHookRule</code>	2715, 2821, 11		
<code>\DeclareHookrule</code>	11		
<code>\def</code>	2648, 2649, 2801, 2802, 2803,		
	2805, 2807, 2809, 2812, 2815, 2816,		
	2817, 2821, 2822, 2823, 2824, 2825, 104		
<code>\DisableGenericHook</code> . .	2643, 2649, 2805, 4		
<code>\DisableHook</code>	2756, 4		
<code>\DiscardShipoutBox</code>	7		
<code>\document</code>	26		
<code>\documentclass</code>	8		
		E	
else commands:			
	<code>\else:</code>	747, 1002, 1157,	
		1219, 1353, 1360, 1368, 1663, 2168,	
		2195, 2332, 2348, 2395, 2430, 2445	
<code>\emph</code>	2682		
<code>\end</code>	28		
<code>\endcsname</code>	8		
<code>\enddocument</code>	27		
<code>\EndIncludeInRelease</code>			
 575, 622, 639, 643, 665, 682,		
	710, 766, 794, 822, 845, 848, 868,		
	871, 884, 902, 907, 910, 916, 921,		
	924, 959, 992, 1041, 1048, 1064,		
	1071, 1091, 1095, 1146, 1150, 1166,		
	1170, 1198, 1201, 1224, 1228, 1245,		
	1250, 1257, 1261, 1297, 1321, 1397,		
	1422, 1454, 1486, 1574, 1648, 1877,		
	1972, 1977, 1982, 2053, 2059, 2083,		
	2099, 2106, 2111, 2133, 2153, 2182,		
	2200, 2205, 2215, 2219, 2222, 2248,		
	2261, 2271, 2288, 2300, 2312, 2322,		
	2352, 2371, 2434, 2438, 2520, 2523,		
	2632, 2638, 2645, 2650, 2657, 2661,		
	2668, 2672, 2705, 2710, 85, 101,		
	129, 145, 157, 169, 185, 198, 218,		
	228, 248, 263, 281, 287, 307, 325, 329		
<code>\EndModuleRelease</code>	2827		
<code>\ERROR</code>	1649, 1650		
<code>\errorstopmode</code>	12		
exp commands:			
	<code>\exp:w</code>	737, 789, 52	
	<code>\exp_after:wN</code>	736, 788, 1001,	
		1003, 1142, 1307, 1468, 1662, 1771,	
		1802, 1887, 2019, 2147, 2325, 2328,	
		2392, 2856, 56, 61, 378, 379, 424, 52	

\exp_args:Nc	1163, 1176, 2088	\hook_disable:n	2724, 2724
\exp_args:NcV	566	\hook_disable_generic:n	2644, 2729, 2761, 264, 264, 266, 282, 285, 14
\exp_args:Ne	699, 700, 2235, 2236	\hook_gclear_next_code:n	2100, 2100, 2674, 16
\exp_args:Nf	1781	\hook_gput_code:nnn	483, 483, 485, 576, 578, 671, 688, 2654, 15
\exp_args:NNe	1563	\hook_gput_code_with_args:nnn	483, 491, 621, 2656, 15
\exp_args:Nnnv	257	\hook_gput_next_code:nn	678, 693, 2039, 2039, 2041, 2054, 2056, 2665, 50
\exp_args:NNo	1024, 1081, 1644, 2857	\hook_gput_next_code_with_args:nn	2047, 2058, 2667, 15
\exp_args:NNV	1537, 1618	\hook_gremove_code:nn	925, 925, 927, 960, 2676, 16
\exp_args:NNx	1450, 1482	\hook_gset_rule:nnnn	1262, 1262, 2716, 2718, 2721, 16
\exp_args:No	1024, 1081, 1141, 1981, 2037, 2847, 2857	\hook_if_empty:n	2336, 2338, 2353, 2355
\exp_args:Nv	1208, 1916, 1923, 2017	\hook_if_empty:nTF	1793, 1896, 2336, 2722, 16
\exp_args:Nx	542, 430, 472	\hook_if_empty_p:n	1846, 1941, 2336, 16
\exp_end:	737, 789	\hook_log:n	1749, 1749, 2712, 16
\exp_last_unbraced:Ne	1026, 1059, 1083, 1175, 2849	\hook_new:n	833, 2619, 67, 69, 88, 223, 95
\exp_last_unbraced:Nf	1188, 1826	\hook_new_pair:nn	2623, 199, 201, 221, 14
\exp_last_unbraced:NNf	1117	\hook_new_pair_with_args:nn	14
\exp_last_unbraced:NNNo	1975, 381	\hook_new_pair_with_args:nnn	2631, 199, 199, 203, 219, 226, 14
\exp_not:N	1111, 1115, 1133, 1136, 1137, 1139, 1183, 1436, 1437, 1564, 1566, 389	\hook_new_reversed:n	2621, 170, 172, 188, 224, 14
\exp_not:n	545, 1008, 1032, 1130, 1134, 1140, 1141, 1570, 2847, 114, 241	\hook_new_reversed_with_args:nn	2629, 170, 170, 174, 186, 197, 14
\exp_stop_f:	1209, 1350, 1358	\hook_new_with_args:nn	2627, 67, 67, 71, 86, 100, 14
\expanded	43	\hook_provide:n	2724, 2731
\ExplSyntaxOff	2863	\hook_provide_pair:nn	2724, 2745
\ExplSyntaxOn	3	\hook_provide_reversed:n	2724, 2738
F			
fi commands:			
\fi:	749, 1004, 1159, 1182, 1222, 1354, 1362, 1368, 1665, 2020, 2123, 2131, 2148, 2170, 2197, 2329, 2334, 2350, 2397, 2432, 2447	\hook_show:n	1749, 1754, 2711, 16
file commands:			
\g_file_curr_name_str	2696	\hook_use:n	1415, 2112, 2112, 2114, 2134, 2136, 2154, 2156, 2210, 2701, 19
G			
\g_@@\meta_{hook}_code_prop	32	\hook_use:nnw	2152, 2181, 2183, 2183, 2185, 2201, 2203, 2211, 2703, 15
\g_@@\meta_{hook}_declared_tl	32	\hook_use_once:n	2249, 2251, 2264, 2702, 93
\g_@@\meta_{hook}_parameter_tl	32	\hook_use_once:nnw	2249, 2249, 2256, 2262, 2269, 2704, 15
\g_@@\meta_{hook}_reversed_tl	32	hook internal commands:	
group commands:			
\group_begin:	1110, 1122, 385	\c__hook_	1254, 1258
\group_end:	1114, 1132, 388	\g__hook_??_code_prop	1251
H			
hook commands:			
\hook_activate_generic:n	2642, 2736, 2755, 2768, 288, 288, 290, 308, 326, 328, 41	\c__hook_??_parameter_tl	1251
\hook_debug_off:	2714, 7, 13, 16	\g__hook_??_reversed_tl	1251
\hook_debug_on:	2713, 7, 8, 16	\g__hook_{hook}_code_prop	32
		\g__hook_{hook}_labels_clist	36
		\g__hook_{hook}_reversed_tl	32

_hook_activate_generic:n	288	_hook_cs_gput_right_slow:nnn	..	993 , 1003 , 1009 , 1046
_hook_activate_generic:nn	2753 , 291 , 292 , 310	_hook_cs_if_empty:N	1151 , 1153 , 1167 , 1169
_hook_activate_generic_pair:nn	2724 , 2750 , 2754 , 2782	_hook_cs_if_empty:NTF	...	1151 , 1816 , 1823 , 2077 , 2079 , 2344 , 2345
_hook_activate_generic_-reversed:n	2724 , 2743 , 2752 , 2755 , 2775	_hook_cs_if_empty_p:N	1151
\g_hook_all_seq	1378 , 1404 , 2854 , 28 , 108 , 137	_hook_cs_parameter_count:N	...	1171 , 1176 , 1186
_hook_apply_rule->:nnn	...	1724	_hook_cs_parameter_count:w	...	1171 , 1188 , 1195 , 1196
_hook_apply_rule<-:nnn	...	1724	\l_hook_cur_hook_tl	1500 , 1581 , 1708 , 1719 , 29 , 79
_hook_apply_rule<:nnn	1724	_hook_curr_name_pop:	2692 , 2796 , 415 , 447 , 45
_hook_apply_rule>:nnn	1724	_hook_curr_name_push:n	2690 , 2695 , 415 , 429 , 45
_hook_apply_rule_xE:nnn	...	1724	_hook_curr_name_push_aux:n	...	415 , 430 , 431
_hook_apply_rule_xW:nnn	...	1724	_hook_currname_or_default:	...	534 , 605 , 2534 , 2578 , 333 , 341 , 345 , 361 , 362 , 362 , 42
_hook_apply_label_pair:nnn	...	1514 , 1515 , 1595 , 1596 , 1651 , 1651 , 86	_hook_debug:n	528, 599, 1377, 1386, 1403, 1406, 1427, 1449, 1459, 1481, 1519, 1539, 1600, 1620, 1676, 1683, 1690, 1698, 1704, 1715, 7, 7, 20, 29	
_hook_apply_rule:nnn	1661 , 1667 , 1667 , 78	_hook_debug_bool	6 , 10 , 15 , 21
_hook_apply_rule:nnnN	78	_hook_debug_gset:	7 , 11 , 16 , 18
_hook_apply_rule->:nnn	1702	_hook_debug_label_data:N	1519 , 1560 , 1600 , 1641 , 1737 , 1737
_hook_apply_rule<-:nnn	1702	_hook_debug_print_rules:n	2023 , 2023
_hook_apply_rule<:nnn	1674	_hook_declare_deprecated-generic:NNn	736 , 757 , 788
_hook_apply_rule>:nnn	1674	_hook_declare_deprecated-generic:NNw	752 , 758 , 759
_hook_apply_rule_xE:nnn	1688	_hook_deprecated_generic-warn:n	735 , 742 , 787 , 953 , 986 , 1273 , 1304 , 1769 , 1885 , 52
_hook_apply_rule_xW:nnn	1688	_hook_deprecated_generic-warn:Nn	742
_hook_braced_cs_parameter:n	...	1027 , 1084 , 1109 , 1119 , 1171 , 1171 , 1173 , 1199 , 2850	_hook_deprecated_generic-warn:Nw	742
_hook_braced_hidden_loop:w	...	1171 , 1175 , 1178 , 1184	_hook_deprecated_generic-warn:w	743 , 744
_hook_braced_parameter:n	1202 , 1202 , 1204 , 1225 , 1227 , 1436 , 1437 , 1564 , 1566	_hook_deprecated_warn:nn	2726 , 2733 , 2740 , 2747 , 2758 , 2765 , 2772 , 2779 , 2784 , 2784
_hook_braced_real_loop:w	...	1202	_hook_disable:n	264 , 267 , 268
_hook_chk_args_allowed:nn	499 , 623 , 623 , 625 , 640 , 642 , 2076 , 87	_hook_do_deprecated_generic:Nn	752 , 752 , 954 , 987 , 1274 , 1305 , 1770 , 1886
_hook_clear_next:n	..	2080 , 2096 , 2101 , 2102 , 2102 , 2104 , 2107 , 2109 , 85	_hook_do_deprecated_generic:Nw	752 , 753 , 754
_hook_clist_gput:Nn	1443, 1445,	1475 , 1477 , 1538 , 1619 , 1649 , 1650			
_hook_code_gset:nn	..	1049 , 1049 , 1051 , 1063 , 1065 , 1067 , 1434 , 2307 , 118			
_hook_code_gset_aux:nnn	..	1012 , 1049 , 1052 , 1054 , 1056 , 1057 , 1070			
_hook_code_gset_auxi:nnnn	993 , 1013 , 1038 , 1040 , 1047 , 1078 , 59			
_hook_cs_end:w	1171 , 1190 , 1191 , 1192 , 1197			
_hook_cs_gput_right:nnn	..	537 , 993 , 993 , 995 , 1042 , 1044 , 2081 , 2279			
_hook_cs_gput_right_fast:nnn	..	993 , 1001 , 1007 , 1045			

```

\__hook_double_hashes:n .....
..... 546, 1024, 1033, 1081, 1137, 63
\c__hook_empty_tl ..... 1243, 35
\__hook_end_document_label_-
check: ..... 415, 454, 455, 462
\__hook_file_hook_normalize:n ...
.... 700, 869, 869, 872, 874, 2236, 55
\l__hook_front_tl .....
... 1489, 1530, 1533, 1536, 1538,
1539, 1540, 1553, 1554, 1611, 1614,
1617, 1619, 1620, 1621, 1634, 1635
\c__hook_generic_⟨type⟩/./⟨place⟩_tl
..... 52
\c__hook_generic_class/./after_-
tl ..... 885
\c__hook_generic_class/./before_-
tl ..... 885
\c__hook_generic_cmd/./after_tl 885
\c__hook_generic_cmd/./before_tl 885
\c__hook_generic_env/./after_tl 885
\c__hook_generic_env/./before_tl 885
\c__hook_generic_env/./begin_tl 885
\c__hook_generic_env/./end_tl .. 885
\c__hook_generic_file/./after_tl 885
\c__hook_generic_file/./before_-
tl ..... 885
\c__hook_generic_include/./after_-
tl ..... 885
\c__hook_generic_include/./before_-
tl ..... 885
\c__hook_generic_include/./end_-
tl ..... 885
\c__hook_generic_package/./after_-
tl ..... 885
\c__hook_generic_package/./before_-
tl ..... 885
\__hook_generic_parameter:n ....
..... 1019, 1238, 1249
\__hook_generic_parameter:w ....
..... 1239, 1240
\c__hook_generics_file_prop 861, 908
\c__hook_generics_prop .....
..... 803, 831, 885, 903, 905
\c__hook_generics_reversed_ii_-
prop ..... 811, 834, 908
\c__hook_generics_reversed_iii_-
prop ..... 814, 837, 908
\__hook_gput_code:nnn .....
.... 483, 488, 494, 497, 579, 580, 654
\__hook_gput_code_store:nnn ....
..... 483, 510, 513
\__hook_gput_next_code:nn .....
... 661, 2044, 2050, 2057, 2060, 2060

\__hook_gput_next_do:nn .....
..... 662, 679, 693,
2066, 2071, 2071, 2073, 2084, 2086, 50
\__hook_gput_next_do:Nnn . 2088, 2091
\__hook_gput_undeclared_hook:nnn
..... 644, 644, 655, 672, 688, 50
\__hook_gremove_code:nn .....
..... 925, 928, 929, 954, 962, 987
\__hook_gset_rule:nnnn 1262, 1264,
1267, 1269, 1274, 1298, 1300, 1305
\c__hook_hash_tl ..... 1183, 63
\c__hook_hashes_tl ..... 1111, 63
\g__hook_hook_curr_name_tl ....
..... 481, 32, 364, 374,
415, 427, 442, 443, 450, 460, 461, 103
\__hook_hook_gput_code_do:nnn ...
483, 517, 526, 587, 597, 647, 240, 257
\__hook_if_cmd_hook:n 2419, 2421, 2435
\__hook_if_cmd_hook:nTF .....
..... 1988, 2419, 2437
\__hook_if_cmd_hook:w ... 2422, 2423
\__hook_if_cmd_hook:wTF ..... 2419
\__hook_if_cmd_hook:p:n ..... 2419
\__hook_if_cmd_hook:p:w ..... 2419
\__hook_if_declared:n ..... 2384
\__hook_if_declared:nTF .....
.. 629, 833, 998, 1015, 1987, 2384,
75, 93, 178, 207, 210, 297, 315, 33
\__hook_if_declared:p:n ..... 2384
\__hook_if_deprecated_generic:n 2407
\__hook_if_deprecated_generic:nTF
..... 733, 785,
951, 984, 1271, 1302, 1767, 1883, 2399
\__hook_if_deprecated_generic:w .
..... 2408, 2409
\__hook_if_deprecated_generic_-
p:n ..... 2399
\__hook_if_disabled:n ..... 273
\__hook_if_disabled:nTF .....
..... 521, 591, 1779, 1791, 1872,
1894, 1967, 2062, 264, 294, 312, 33
\__hook_if_disabled_p:n ..... 264
\__hook_if_execute_immediately:n
..... 2323
\__hook_if_execute_immediately:nTF
500, 582, 1278, 2253, 2258, 2266, 2323
\__hook_if_execute_immediately_-
p:n ..... 2323
\__hook_if_file_hook:w . 846, 849, 851
\__hook_if_file_hook:wTF .....
..... 697, 846, 2233, 91
\__hook_if_file_hook_p:w ..... 846
\__hook_if_generic:n ..... 2399

```



```

__hook_if_generic:nTF .....
.... 716, 772, 1018, 1777, 1891, 2399
__hook_if_generic:w ..... 2400, 2401
__hook_if_generic_p:n ..... 2399
__hook_if_generic_reversed:n . 2439
__hook_if_generic_reversed:nTF .
..... 728, 780, 2439, 302, 320
__hook_if_generic_reversed:w ...
..... 2440, 2441
__hook_if_generic_reversed_p:n 2439
__hook_if_label_case:nnnn .....
..... 1364, 1364, 1512, 1593, 2001
__hook_if_replacing_args: ... 2453
__hook_if_replacing_args:TF ...
.. 502, 544, 555, 627, 1031, 2449,
2455, 2460, 2461, 2466, 2467, 2472
__hook_if_reversed:n ..... 2390
__hook_if_reversed:nTF .....
..... 1441, 1473, 1812,
1852, 1854, 1910, 1947, 1949, 2390
__hook_if_reversed_p:n ..... 2390
__hook_if_structure_exist:n . 2378
__hook_if_structure_exist:nTF ..
..... 931, 964,
2065, 2357, 2378, 2609, 150, 162, 34
__hook_if_structure_exist_p:n 2378
__hook_if_usable:n ..... 2372
__hook_if_usable:nTF 515, 529, 557,
585, 600, 718, 774, 805, 947, 980,
1430, 1462, 1789, 1811, 1870, 1892,
1909, 1965, 2281, 2372, 2723, 106, 57
__hook_if_usable_p:n 1845, 1940, 2372
__hook_if_usable_use:n .....
..... 2220, 2235, 2238, 2240, 91
__hook_include_legacy_code_-
chunk:n ..... 1429,
1461, 126, 142, 229, 229, 231, 249, 251
__hook_init_structure:n .....
.. 536, 607, 646, 1284, 1309, 2075,
119, 139, 146, 146, 148, 158, 160, 37
__hook_initialize_all: ... 1372,
1372, 1374, 1398, 1400, 1420, 2794
__hook_initialize_hook_code:n ..
..... 1376, 1402, 1423, 1423,
1425, 1455, 1457, 2130, 2151, 2179, 77
__hook_initialize_single:NNn ...
..... 1447, 1479,
1494, 1494, 1496, 1573, 1575, 1577, 69
\l_hook_label_0_tl ..... 1489
__hook_label_if_exist_apply:nnnTF
..... 1651, 1653, 1655, 1658, 78
__hook_label_ordered:nn ... 1356, 69
__hook_label_ordered:nnTF .....
..... 1325, 1331, 1337, 1356, 68
__hook_label_ordered_p:nn ... 1356
__hook_label_pair:nn .....
..... 1324, 1330, 1336, 1340,
1343, 1347, 1348, 1348, 1733, 1734, 69
\l_hook_labels_int .....
..... 1489, 1499, 1503,
1535, 1556, 1580, 1584, 1616, 1637, 75
\l_hook_labels_seq .. 1489, 1498,
1504, 1522, 1579, 1585, 1603, 1739
__hook_list_if_rule_exists:nnnTF
..... 1994, 2011, 2012, 2014
__hook_list_one_rule:nnn .....
..... 1994, 2003, 2004, 2009
__hook_list_rules:nn .....
.... 1832, 1927, 1994, 1994, 2028, 85
__hook_log:nN .. 1749, 1752, 1757,
1763, 1765, 1770, 1879, 1881, 1886
__hook_log_cmd:n .....
.. 1751, 1756, 1760, 1762, 1774, 1890
__hook_log_line:n .....
1749, 1759, 1790, 1792, 1796, 1808,
1820, 1830, 1848, 1867, 1893, 1895,
1899, 1906, 1918, 1925, 1943, 1962
__hook_log_line_indent:n .....
..... 1749, 1761,
1798, 1804, 1814, 1821, 1835, 1843,
1901, 1904, 1912, 1919, 1930, 1938
__hook_log_next_code:n .....
..... 1923, 1973, 1973, 1978, 1980
__hook_log_next_code:w .. 1826, 1976
__hook_make_name:n .....
. 355, 361, 370, 376, 376, 430, 472, 43
__hook_make_name:w ... 376, 378, 382
__hook_make_usable:n .....
..... 778, 809, 97, 133, 318
__hook_make_usable:nn .. 723, 726,
81, 102, 102, 104, 130, 132, 300, 36
__hook_misused_if_replacing_-
args:nn ..... 2449, 2450, 2456
__hook_msg_pair_found:nnn .....
..... 1676, 1683,
1690, 1698, 1706, 1717, 1730, 1730
\g_hook_name_stack_seq ..... 32,
416, 417, 421, 428, 442, 449, 457, 467
__hook_new:n ..... 89, 91, 193
__hook_new:nn ..... 67
__hook_new:nn 70, 72, 73, 90, 181, 213
__hook_new_pair:nnn .. 202, 204, 205
__hook_new_reversed:n ..... 189, 191
__hook_new_reversed:nn .....
.... 170, 173, 175, 176, 190, 196, 214
__hook_next_⟨hook⟩ ..... 57
__hook_next_gset:nn ..... 937,
1049, 1055, 1069, 2080, 2105, 2308, 154

```

```

\c__hook_nine_parameters_tl . . . .
    . . . . . 1020, 1134, 1803, 35, 114
\__hook_normalise_code_pool:n . . .
    . . . . . 1096, 1096, 1098, 1147, 1149, 120, 36
\__hook_normalise_cs_args:nn 1072,
    . . . . . 1072, 1074, 1092, 1094, 116, 117, 36
\__hook_normalise_fn:nn . . . . .
    . . . . . 561, 1102, 1120, 1145, 61
\__hook_normalize_hook_args:Nn . .
    . . . . . 1752, 1757, 2044, 2050, 2057,
    . . . . . 2101, 2254, 2259, 2267, 2753, 70,
    . . . . . 72, 89, 173, 175, 189, 267, 291, 383, 392
\__hook_normalize_hook_args:Nnn .
    . . . . . 488, 494, 579, 928, 202, 204, 383, 397
\__hook_normalize_hook_args-
    aux:Nn . . . . . 383, 383, 394, 399, 407
\__hook_normalize_hook_rule-
    args:Nnnnn . . . . . 1264, 383, 405
\__hook_parameter:n . . 1060, 1118,
    . . . . . 1229, 1229, 1231, 1246, 1248, 1784
\c__hook_parameter_cmd . . . . . 917, 922
\c__hook_parameter_cmd/.after-
    tl . . . . . 917
\c__hook_parameter_cmd/.before-
    tl . . . . . 917
\__hook_parse_dot_label:n . . . . .
    . . . . . 334, 336, 336
\__hook_parse_dot_label:w . . . . .
    . . . . . 336, 346, 349
\__hook_parse_dot_label_aux:w . . .
    . . . . . 336, 352, 360
\__hook_parse_dot_label_cleanup:w
    . . . . . 336, 356, 359
\__hook_parse_label_default:n . . .
    . . . . . 330, 330, 395, 401, 402, 409, 410, 412
\__hook_patch_cmd_or_delay:Nnn . . 52
\__hook_post_initialization-
    defs: . . . . . 1395,
    . . . . . 2206, 2206, 2208, 2213, 2216, 2218
\__hook_preamble_hook:n . . . . .
    . . . . . 1416, 1773, 1889,
    . . . . . 2112, 2116, 2127, 2140, 2150, 2160,
    . . . . . 2178, 2187, 2212, 2244, 2276, 2294, 89
\__hook_print_args:n . . . . . 1983
\__hook_print_args:nn . . . 1781, 1983
\__hook_prop_gput_labeled-
    cleanup:nnn . . . . . 483, 542, 552
\__hook_prop_gput_labeled_do:Nnn
    . . . . . 566, 569
\__hook_prop_gput_labeled-
    do:Nnnn . . . . . 483
\l__hook_rear_tl . . . . .
    . . . . . 1489, 1520, 1526, 1527, 1549,
    . . . . . 1550, 1601, 1607, 1608, 1630, 1631
\__hook_replacement_spec:N 1155, 1161
\__hook_replacing_args_false: . . .
    . . . . . 487, 634,
    . . . . . 1381, 2043, 2278, 2449, 2463, 237, 46
\__hook_replacing_args_reset: . . .
    . . . . . 489, 495,
    . . . . . 1384, 2045, 2051, 2280, 2449, 2469, 243
\__hook_replacing_args_true: . . .
    . . . . . 493, 1382, 2049, 2449, 2457
\g__hook_replacing_stack_seq . 2449
\l__hook_return_tl . . . . .
    . . . . . 554, 562, 567, 571, 572, 613,
    . . . . . 616, 943, 976, 1536, 1537, 1617,
    . . . . . 1618, 2471, 2472, 25, 449, 450, 457, 461
\__hook_rollback_tidyng: . . . . 2834
\__hook_rule<_gset:nnn . . . . . 1322
\__hook_rule>_gset:nnn . . . . . 1322
\__hook_rule_after_gset:nnn . . .
    . . . . . 1322, 1328, 1333
\__hook_rule_before_gset:nnn . . .
    . . . . . 1322, 1322, 1327, 74
\__hook_rule_gclear:nnn . . . . .
    . . . . . 1285, 1310, 1345, 1346, 69
\__hook_rule_incompatible-error-
    gset:nnn . . . . . 1339
\__hook_rule_incompatible-warning-
    gset:nnn . . . . . 1339
\__hook_rule_unrelated_gset:nnn .
    . . . . . 1345, 1345, 69
\__hook_rule_voids_gset:nnn . . .
    . . . . . 1334, 1334
\__hook_seq_csname:n . . . . .
    . . . . . 1487, 1488, 1506,
    . . . . . 1540, 1587, 1621, 1679, 1686, 1744
\__hook_set_default_hook_label:n
    . . . . . 2678, 465, 465
\__hook_set_default_label:n . . .
    . . . . . 465, 472, 474
\__hook_set_normalise_fn:nn . . .
    . . . . . 559, 1096, 1100, 1105, 61
\__hook_str_compare:nn . . . . .
    . . . . . 1350, 1358, 1367, 23, 23
\__hook_strip_double_slash:n . . .
    . . . . . 869, 875, 876
\__hook_strip_double_slash:w . . .
    . . . . . 869, 877, 878, 882
\__hook_tl_csname:n . . . . .
    . . . . . 1487, 1487, 1493, 1505,
    . . . . . 1521, 1524, 1526, 1530, 1542, 1544,
    . . . . . 1547, 1549, 1554, 1586, 1602, 1605,
    . . . . . 1607, 1611, 1623, 1625, 1628, 1630,
    . . . . . 1635, 1677, 1678, 1684, 1685, 1743
\__hook_tl_gclear:N . . . . .
    . . . . . 969, 970, 974, 1531, 1612,

```

2317, 2318, 2319, <u>64</u> , 64, 66, 244, 259	_hook_use_i_delimit_by_s_-
_hook_tl_gput:Nn	mark:nw 1181, <u>45</u> , 46
. 1442, 1444, 1474, 1476,	_hook_use_initialized:n
1537, 1563, 1618, 1644, <u>1649</u> , 1649, 76 1415, <u>2112</u> , 2117,
_hook_tl_gput_left:Nn	2119, 2144, 2165, 2210, 2282, 2296, 93
. 1442, 1474, <u>58</u> , 58	_hook_use_initialized:nnw
_hook_tl_gput_right:Nn 2183, 2188, 2190, 2211
. 608, 1444, 1476,	_hook_use_none_delimit_by_s_-
1565, 1645, 1647, 2097, <u>55</u> , 55, 57	mark:w 1276, 1282, 2325, 2392, <u>45</u> , 45
_hook_tl_gset:Nn 1324,	_hook_use_once:n 2267, 2292
1330, 1336, 1340, 1343, 1466, 2096,	_hook_use_once:nn 2254,
2304, 2845, <u>49</u> , 49, 51, 53, 54, 56, 60	2259, 2272, 2272, 2274, 2290, 2299
_hook_tl_gset_eq:NN <u>63</u> , 63, 65	_hook_use_once_clear:n 2279,
_hook_tl_set:Nn	2297, <u>2301</u> , 2301, 2305, 2313, 2315, 93
. 1111, 1505, 1586, <u>47</u> , 47, 31	_hook_use_once_set:n
_hook_tmp:w 2277, 2295, <u>2301</u> , 2303, 93
. 1107, 1118, 1133, 1139, 1142,	_hook_use_undefined:w 2169, 2173
1802, 1805, 1996, 2017, 2026, 2037,	\\g_hook_used_prop
2843, 2859, 2860, <u>34</u> , 34, 418, 422, 424	1377, 1389, 1403, 1409, 1450, 1482, <u>31</u>
\\l_hook_tmpa_bool 1831, 1834, 1842,	\\l_hook_work_prop 562, 1101, 1103,
1851, 1926, 1929, 1937, 1946, <u>24</u> , 82	1136, 1446, 1478, 1501, 1508, 1510,
\\l_hook_tmpa_tl 1119, 1140, <u>25</u> , 428	1519, 1536, 1560, 1582, 1589, 1591,
\\l_hook_tmpb_tl 1108, 1115, 1142, <u>25</u>	1600, 1617, 1641, 1711, 1722, <u>30</u> , 74
_hook_toplevel_⟨hook⟩ 57	hook_?? internal commands:
_hook_toplevel_gset:nn	_hook_?? 66
936, 941, <u>1049</u> , 1053, 1068, 2309, 153	hook_⟨hook⟩ internal commands:
_hook_try_declaring_generic_-	_hook_⟨hook⟩ 32
hook:nnn 523, 593,	hook ?? internal commands:
<u>649</u> , 649, 651, 666, 668, 683, 685, 50	_hook~?? <u>1251</u>
_hook_try_declaring_generic_-	
hook:nNNnn 687, 692, <u>695</u> , 695, 51	I
_hook_try_declaring_generic_-	if commands:
hook:wn 711,	\\if:w 997,
713, 767, 769, 795, 797, 823, 825	1155, 1180, 2326, 2340, 2393, 2425, 94
_hook_try_declaring_generic_-	\\if_case:w 1206, 1350, 1367
hook:wnTF 653,	\\if_charcode:w 2443
660, 670, 677, 706, <u>711</u> , 762, 52	\\if_cs_exist:w 747, 1660, 2016,
_hook_try_declaring_generic_-	2121, 2129, 2146, 2167, 2193, 2327
hook_split:nNNnn <u>695</u> , 699, 702, 704	\\if_int_compare:w 1358
_hook_try_declaring_generic_-	\\IfHookEmptyTF <u>2722</u> , 2825, 104
next_hook:nn	\\IfHookExistsTF <u>2723</u> , 2824, 104
. <u>649</u> , 658, 675, 690, 2067, 50	\\ignorespaces 26
_hook_try_file_hook:n	\\immediate 27
. <u>2220</u> , 2228, 2231, 91	\\include 25
_hook_try_put_cmd_hook:n	\\IncludeInRelease
. 722, 777, 808	. 483, 576, 623, 640, 649, 666, 683,
_hook_update_hook_code:n	711, 767, 795, 823, 846, 849, 869,
. 518, 588, 948, 981, 1289,	872, 885, 903, 908, 911, 917, 922,
1314, <u>1371</u> , 1371, 1376, 1383, 1402,	925, 960, 993, 1042, 1049, 1065,
1405, 2078, 2094, 82, 303, 321, 35	1072, 1092, 1096, 1147, 1151, 1167,
_hook_use:wn	1171, 1199, 1202, 1225, 1229, 1246,
<u>2163</u> , 2176, <u>2220</u> , 2220, 2223, 2225, 91	1254, 1258, 1267, 1298, 1372, 1398,
_hook_use_end: 2171, 2173, 2180	1423, 1455, 1494, 1575, 1763, 1879,
	1973, 1978, 2039, 2054, 2071, 2084,

2102, 2107, 2112, 2134, 2154, 2183, 2201, 2206, 2216, 2220, 2223, 2249, 2262, 2272, 2290, 2301, 2313, 2336, 2353, 2419, 2435, 2492, 2521, 2624, 2633, 2639, 2646, 2651, 2658, 2662, 2669, 2699, 2706, 2798, 67, 86, 102, 130, 146, 158, 170, 186, 199, 219, 229, 249, 264, 282, 288, 308, 326	\msg_expandable_error:nn 340 \msg_expandable_error:nnn 1220, 2452, 368 \msg_line_context: 560, 2533, 2538, 2585 \g_msg_module_name_prop 2476 \g_msg_module_type_prop . . 2474, 2475 \msg_new:nnn 2531, 2536, 2581, 2587, 2592, 2597, 2602, 2606, 2613, 2786 \msg_new:nnnn . . . 2477, 2487, 2494, 2501, 2510, 2524, 2541, 2557, 2570 \msg_warning:nnn 295, 313 \msg_warning:nnnn 944, 956, 977, 989, 2785 \msg_warning:nnnnn 748 \msg_warning:nnnnnn 1699
\input 9 \InsertMark 29 int commands: \int_compare:nNnTF 1524, 1546, 1556, 1605, 1627, 1637, 1985, 1991, 2284, 2481 \int_compare_p:nNn 2831, 2832 \int_decr:N 1535, 1616 \int_eval:n 1207, 1543, 1624, 1678, 1685, 1783 \int_incr:N 1503, 1584 \int_new:N 1490 \int_set:Nn 1123 \int_zero:N 1499, 1580 iow commands: \iow_char:N 1708, 1719, 2504, 2506, 2508, 2513, 2516, 2518, 2528, 2564, 2572, 2584, 2589, 2594, 2788, 2790 \iow_log:n 1751 \iow_term:n 528, 599, 1388, 1391, 1408, 1411, 1428, 1460, 1539, 1558, 1559, 1561, 1620, 1639, 1640, 1642, 1707, 1718, 1732, 1738, 1739, 1740, 1743, 1747, 1756, 2025, 2030	N \NewDocumentCommand 2618, 2620, 2622, 2626, 2628, 2630, 2641, 2643, 2653, 2655, 2664, 2666, 2673, 2675, 2677, 2689, 2691, 2715, 2717, 2720 \newenvironment 24 \NewHook 2618, 2801, 2802 \NewHookWithArguments 2624, 3 \NewMirroredHookPair 2618, 2803, 3 \NewMirroredHookPairWithArguments 2624, 3 \NewModuleRelease 4 \NewReversedHook 2618, 2802, 3 \NewReversedHookWithArguments . . 2624, 3 \normalfont 28 \normalsize 7
K kernel internal commands: _kernel_cs_parm_from_arg_- count:nnTF 109 _kernel_exp_not:w . . . 48, 50, 56, 61	O or commands: \or: . . 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1352, 1368
L \let 2818, 2819, 104 \listfiles 27 \LogHook 2711, 12 \long 2807, 2824, 2825	P \PopDefaultHookLabel 2677, 10 prg commands: \prg_do_nothing: 2213 \prg_new_conditional:Npnn 851, 1153, 1356, 2323, 2338, 2355, 2372, 2378, 2384, 2390, 2399, 2407, 2421, 2439, 273 \prg_new_protected_conditional:Npnn 713, 769, 797, 825 \prg_replicate:nn 1127, 2196, 2204, 2270, 2285 \prg_return_false: 739, 791, 801, 819, 829, 842, 859, 863, 866, 1158, 1361, 2333,
M \mdseries 28 msg commands: \msg_error:nn 434, 451 \msg_error:nnn 522, 539, 592, 610, 2063, 76, 94, 179, 208, 211, 459 \msg_error:nnnn 504, 633, 112 \msg_error:nnnnn 438, 469, 478 \msg_error:nnnnnn 1280, 1292, 1317, 1691	

2349, 2367, 2376, 2382, 2388, 2396, 2405, 2414, 2417, 2431, 2446, 279, 52	\scan_stop: 697, . 653, 660, 670, 677, 706, 714, 750, 762, 770, 798, 826, 1155, 2440, 2441
\prg_return_true: 730, 782, 817, 840, 862, 1156, 1359, 2331, 2347, 2366, 2369, 2375, 2381, 2387, 2394, 2404, 2415, 2429, 2444, 278	scan internal commands: \s__hook_mark 697, 743, 745, 753, 755, 758, 760, 852, 877, 878, 882, 1176, 1193, 1197, 1239, 1240, 1295, 2163, 2176, 2225, 2233, 2331, 2333, 2394, 2396, 2400, 2401, 2408, 2409, 2422, 2423, 44, 45, 46, 346, 349, 352, 356, 359, 360, 31
prop commands: \prop_clear:N 1101 \prop_const_from_keyval:Nn 905, 913, 914, 915 \prop_gclear:N 935, 968, 1377, 1403 \prop_gclear_new:N 2310, 2320 \prop_get:NnN 562, 1536, 1617 \prop_get:NnNTF 571, 613 \prop_gpop:NnNTF 943, 976 \prop_gput:Nnn 572, 573, 615, 618, 1450, 1482, 2474, 2475, 2476 \prop_gset_eq:NN 1103 \prop_if_empty:NTF 1432, 1464, 1797, 1900, 2343 \prop_if_empty_p:N 2360 \prop_if_exist:NTF 2342, 2380 \prop_if_in:NnTF 803, 811, 814, 831, 834, 837, 861 \prop_map_break: 1513, 1594, 2002 \prop_map_function:NN 1102, 62 \prop_map_inline:Nn 1389, 1409, 1501, 1508, 1510, 1582, 1589, 1591, 1741, 1800, 1903, 1997, 1999 \prop_new:N 1251, 1252, 30, 31, 152, 164 \prop_put:Nnn 1136, 1711, 1722, 37, 42 \prop_set_eq:NN 1446, 1478 \prop_show:N 86 \providecommand 2810, 2813 \ProvideHook 2756 \ProvideMirroredHookPair 2756 \ProvideReversedHook 2756 \PushDefaultHookLabel 2677, 8	\selectfont 28 seq commands: \seq_clear:N 1498, 1579 \seq_clear_new:N 1506, 1587 \seq_gpop:NN 2471 \seq_gpop:NNTF 449, 457 \seq_gpop_right:NN 428 \seq_gpush:Nn 2459, 2465, 442 \seq_gput_right:Nn 108, 137, 417, 421 \seq_if_empty:NTF 416, 467 \seq_map_inline:Nn 1378, 1404, 1522, 1540, 1603, 1621, 2854 \seq_new:N 1489, 2449, 28, 33 \seq_put_right:Nn 1504, 1585, 1679, 1686 \seq_use:Nnnn 1739, 1744 \SetDefaultHookLabel 2677, 10 \sffamily 28 \shipout 28 show commands: \show_hook:n 80 \ShowHook 2711, 2817, 19 \small 7 \sourceLaTeXdate 2831 \space 531, 602 \special 27 str commands: \c_right_brace_str 1976 \str_count:n 1208, 1784, 37, 43 \str_gset:Nn 2696 \str_if_eq:nn 69 \str_if_eq:nnTF 532, 534, 603, 605, 720, 776, 807, 854, 933, 940, 966, 973, 1732, 1838, 1933, 2033, 2227, 2426, 2480, 2545, 344, 436, 476 \str_if_eq_p:nn 858, 1533, 1614 str internal commands: __str_if_eq:nn 23, 30 \strut 29
Q	
quark commands: \quark_if_recursion_tail_stop:n 420 \q_recursion_stop 426 \q_recursion_tail 425, 426	
R	
\RemoveFromHook 2675, 2812, 12 \requestedLaTeXdate 2832 \RequirePackage 25 \rmfamily 28	
S	
scan commands: \scan_new:N 44	
T	
TeX and L ^A T _E X 2 _ε commands: \@...hook 39	

