

The `lthooks` package*

Frank Mittelbach†

January 9, 2021

Contents

1	Introduction	2
2	Package writer interface	2
2.1	L ^A T _E X 2 _ε interfaces	2
2.1.1	Declaring hooks and using them in code	2
2.1.2	Updating code for hooks	3
2.1.3	Hook names and default labels	5
2.1.4	The <code>top-level</code> label	7
2.1.5	Defining relations between hook code	7
2.1.6	Querying hooks	8
2.1.7	Displaying hook code	9
2.1.8	Debugging hook code	11
2.2	L ³ programming layer (<code>exp13</code>) interfaces	11
2.3	On the order of hook code execution	13
2.4	The use of “reversed” hooks	15
2.5	Difference between “normal” and “one-time” hooks	16
2.6	Private L ^A T _E X kernel hooks	16
2.7	Legacy L ^A T _E X 2 _ε interfaces	17
2.8	L ^A T _E X 2 _ε commands and environments augmented by hooks	18
2.8.1	Generic hooks for all environments	18
2.8.2	Hooks provided by <code>\begin{document}</code>	19
2.8.3	Hooks provided by <code>\end{document}</code>	19
2.8.4	Hooks provided <code>\shipout</code> operations	21
2.8.5	Hooks provided by file loading operations	21
2.8.6	Hooks provided in NFSS commands	21
	Index	22

*This package has version v1.0h dated 2021/01/07, © L^AT_EX Project.

†Code improvements for speed and other goodies by Phelype Oleinik

1 Introduction

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

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

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

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

2 Package writer interface

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

2.1 $\text{\LaTeX} 2_{\epsilon}$ interfaces

2.1.1 Declaring hooks and using them in code

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

`\NewHook`

`\NewHook {<hook>}`

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

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section [2.1.3](#).

`\NewReversedHook`

`\NewReversedHook {<hook>}`

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

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section [2.1.3](#).

`\NewMirroredHookPair` `\NewMirroredHookPair {<hook-1>} {<hook-2>}`

A shorthand for `\NewHook{<hook-1>}\NewReversedHook{<hook-2>}`.

The `<hooks>` can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\UseHook` `\UseHook {<hook>}`

Execute the hook code inside a command or environment.

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

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

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

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

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

FMi: Maybe add an error version as well?

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

2.1.2 Updating code for hooks

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

Adds `<code>` to the `<hook>` labeled by `<label>`. When the optional argument `<label>` is not provided, the `<default label>` is used (see section 2.1.3). 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 2.1.4).

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 2.1.6.

The `<hook>` and `<label>` can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

\RemoveFromHook

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

Removes any code labeled by `<label>` from the `<hook>`. When the optional argument `<label>` is not provided, the `<default label>` is used (see section 2.1.3).

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

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

The `<hook>` and `<label>` can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

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 to font size changes for no good reason.

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

\AddToHookNext

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

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

Using the declaration is a global operation, i.e., the code is not lost, even if the declaration is used inside a group and the next invocation happens after the group. If the declaration is used several times before the hook is executed then all code is executed in the order in which it was declared.¹

It is possible to nest declarations 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 2.1.6.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

2.1.3 Hook names and default labels

It is best practice to use `\AddToHook` in packages or classes *without specifying a $\langle label \rangle$* 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`, `\IfHookEmptyTF`, and `\IfHookExistsTF` (and their `expl3` interfaces `\hook_use:n`, `\hook_use_once:n`, `\hook_if_empty:nTF`, and `\hook_if_exist: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`, `\IfHookEmptyTF`, and `\IfHookExistsTF` 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/after/foo.tex}{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/after/foo.tex}{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`,

¹There is no mechanism to reorder such code chunks (or delete them).

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 L^AT_EX 2_ε interface, and the L^AT_EX 3 interface described in section 2.2.

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`) and `\IfHookExistsTF` (and `\hook_if_exist:nTF`) because these conditionals are used in some performance-critical parts of the hook management code, and because they are usually used to refer to other package’s hooks, so the dot-syntax doesn’t make much sense.

In some cases, for example in large packages, one may want to separate it in logical parts, but still use the main package name as $\langle label \rangle$, then the $\langle default label \rangle$ can be set using `\SetDefaultHookLabel` or `\PushDefaultHookLabel.. \PopDefaultHookLabel`.

```
\PushDefaultHookLabel
\PopDefaultHookLabel
```

```
\PushDefaultHookLabel { $\langle default label \rangle$ }
 $\langle code \rangle$ 
\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 2.1.4) 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 emulate `\usepackage`-like hook behaviour 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`.

\SetDefaultHookLabel**\SetDefaultHookLabel** {*default label*}

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

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

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

2.1.4 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.

2.1.5 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.

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

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

Currently, the supported relations are the following:

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

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

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

`incompatible-error` Like `incompatible-error` but instead of a warning a 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 2.1.3.

`\ClearHookRule` `\ClearHookRule{<hook>}{<label1>}{<label2>}`

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

`\DeclareDefaultHookRule` `\DeclareDefaultHookRule{<label1>}{<relation>}{<label2>}`

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

Declaring default rules is only supported in the document preamble.²

The `<label>` can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

2.1.6 Querying hooks

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

- exist and be empty;

²Trying 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: they have four possible states. A hook may exist or not, and either way it may or may not be empty. This means that even a hook that doesn't exist may be non-empty.

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

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

```
\IfHookEmptyTF * \IfHookEmptyTF <hook> <true code> <false code>
```

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

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

```
\IfHookExistsTF * \IfHookExistsTF <hook> <true code> <false code>
```

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

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

Generic hooks are declared at the time code is added to them, so the result of `\hook_if_exist:n` will change once code is added to said hook (unless the hook was previously declared).

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

FMi: Would be helpful if we provide some use cases

2.1.7 Displaying hook code

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

`\ShowHook`
`\LogHook`

`\ShowHook {⟨hook⟩}`

Displays information about the *⟨hook⟩* such as

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

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

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

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

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

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

`⟨label⟩ -> ⟨code⟩`

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

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

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

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

`-> ⟨next-code⟩`

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

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

`<label-1>|<label-2> with <default?> relation <relation>`

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

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

2.1.8 Debugging hook code

```
\DebugHooksOn
\DebugHooksOff
```

```
\DebugHooksOn
```

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

2.2 L3 programming layer (expl3) interfaces

This is a quick summary of the 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 `<label>` for a code chunk. We therefore suggest to use the declarations discussed in the previous section even in `expl3` packages, but the choice is yours.

```
\hook_new:n
\hook_new_reversed:n
\hook_new_pair:nn
```

```
\hook_new:n {<hook>}
\hook_new_reversed:n {<hook>}
\hook_new_pair:nn {<hook-1>} {<hook-2>}
```

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

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section [2.1.3](#).

```
\hook_use:n
```

```
\hook_use:n {<hook>}
```

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

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

```
\hook_use_once:n
```

```
\hook_use_once:n {<hook>}
```

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

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

`\hook_gput_code:nnn`

`\hook_gput_code:nnn` $\langle hook \rangle$ $\langle label \rangle$ $\langle code \rangle$

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

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

The $\langle hook \rangle$ and $\langle label \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\hook_gput_next_code:nn`

`\hook_gput_next_code:nn` $\langle hook \rangle$ $\langle code \rangle$

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

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

Thus if one needs to undo what the standard does one has to do that as part of $\langle code \rangle$.

The $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\hook_gremove_code:nn`

`\hook_gremove_code:nn` $\langle hook \rangle$ $\langle label \rangle$

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

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

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

The $\langle hook \rangle$ and $\langle label \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\hook_gset_rule:nnnn`

`\hook_gset_rule:nnnn` $\langle hook \rangle$ $\langle label1 \rangle$ $\langle relation \rangle$ $\langle label2 \rangle$

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

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

`\hook_if_empty_p:n` \star

`\hook_if_empty:nTF` $\langle hook \rangle$ $\langle true code \rangle$ $\langle false code \rangle$

`\hook_if_empty:nTF` \star

Tests if the $\langle hook \rangle$ is empty (i.e., no code was added to it using either `\AddToHook` or `\AddToHookNext`), 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.

`\hook_if_exist_p:n` * `\hook_if_exist:nTF` $\langle hook \rangle$ $\langle true\ code \rangle$ $\langle false\ code \rangle$
`\hook_if_exist:nTF` * Tests if the $\langle hook \rangle$ exists (if it was created with either `\NewHook`, `\NewReversedHook`, or `\NewMirroredHookPair`), and branches to either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ depending on the result.

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

Generic hooks are declared at the time code is added to them, so the result of `\hook_if_exist:n` will change once code is added to said hook (unless the hook was previously declared).

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

`\hook_show:n` `\hook_show:n` $\langle hook \rangle$
`\hook_log:n` 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.

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

The $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\hook_debug_on:` `\hook_debug_on:`
`\hook_debug_off:` Turns the debugging of hook code on or off. This displays changes to the hook data.

2.3 On the order of hook code execution

Chunks of code for a $\langle hook \rangle$ 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 example³, suppose there is a package adding the following:

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

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

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

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

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

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

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

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

```
-> The hook 'env/quote/after':
> Code chunks:
>   package-1 -> \end {itshape}
>   package-too -> \end {color}
> Document-level (top-level) code (executed first):
>   ---
> 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.

³there are simpler ways to achieve the same effect.

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

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

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

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

This has some consequences one needs to be aware of:

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

In particular this means that construct such as

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

works for one-time hooks⁴ (all three code chunks are executed one after another), but it makes little sense with a normal hook, because with a normal hook the first time `\UseHook{myhook}` is executed it would

- execute `<code-1>`,
- then execute `\AddToHook{myhook}{code-2}` which adds the code chunk `<code-2>` to the hook for use on the next invocation,
- and finally execute `<code-3>`.

The second time `\UseHook` is called it would execute the above and in addition `<code-2>` as that was added as a code chunk to the hook in the meantime. So each time the hook is used another copy of `<code-2>` is added and so that code chunk is executed $\langle \# \text{ of invocations} \rangle - 1$ times.

2.6 Private \LaTeX kernel hooks

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

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

⁴This is sometimes used with `\AtBeginDocument` which is why it is supported.


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

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

2.7 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.

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

If used without the optional argument *label*, it works essentially like before, i.e., it is adding *code* to the hook `begindocument` (which is executed inside `\begin{document}`). However, all code added this way is labeled with the label `top-level` (see section 2.1.4) if done outside of a package or class or with the package/class name if called inside such a file (see section 2.1.3).

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

`\AtBeginDocument` is a wrapper around the `begindocument` hook (see section 2.8.2), which is a one-time hook. As such, after the `begindocument` hook is executed at `\begin{document}` any attempt to add *code* to this hook with `\AtBeginDocument` or with `\AddToHook` will cause that *code* to execute immediately instead. See section 2.5 for more on one-time hooks.

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

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

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

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

This hook is discussed in conjunction with the shipout hooks.

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.

⁵As with everything in T_EX there is not enforcement of this rule, and by looking at the code it is easy to find out how the kernel adds to them. The main reason of this section is therefore to say “please don’t do that, this is unconfigurable code!”

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

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

intro to be written

2.8.1 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 $\backslash begin$ 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 $\backslash begin$ directly in front of the code specific to the environment start (e.g., the second argument of $\backslash newenvironment$). Its scope is the environment body.

env/ $\langle env \rangle$ /end This hook is executed as part of $\backslash end$ directly in front of the code specific to the end of the environment (e.g., the third argument of $\backslash newenvironment$).

env/ $\langle env \rangle$ /after This hook is executed as part of $\backslash end$ 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.⁶ In contrast to other hooks there is also no need to declare them using $\backslash NewHook$.

The hooks are only executed if $\backslash begin\{\langle env \rangle\}$ and $\backslash end\{\langle env \rangle\}$ is used. If the environment code is executed via low-level calls to $\backslash \langle env \rangle$ and $\backslash end\langle env \rangle$ (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.

$\backslash BeforeBeginEnvironment$ [$\langle label \rangle$] [$\langle code \rangle$]

This declaration adds to the **env/ $\langle env \rangle$ /before** hook using the $\langle label \rangle$. If $\langle label \rangle$ is not given, the $\langle default label \rangle$ is used (see section 2.1.3).

$\backslash AtBeginEnvironment$ [$\langle label \rangle$] [$\langle code \rangle$]

Like $\backslash BeforeBeginEnvironment$ but adds to the **env/ $\langle env \rangle$ /begin** hook.

⁶Thus 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.

\AtEndEnvironment**\AtEndEnvironment** [*label*] {*code*}Like **\BeforeBeginEnvironment** but adds to the **env/⟨env⟩/end** hook.

\AfterEndEnvironment**\AfterEndEnvironment** [*label*] {*code*}Like **\BeforeBeginEnvironment** but adds to the **env/⟨env⟩/after** hook.

2.8.2 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):

begin{document}/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).

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

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

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

begin{document}/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.

2.8.3 Hooks provided by **\end{document}**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

There is also the hook `shipout/lastpage`. This hook is executed as part of the last `\shipout` in the document to allow package to add final `\special`'s to that page. Where this hook is executed in relation to those from the above list can vary from document to document. Furthermore to determine correctly which of the `\shipouts` is the last one, \LaTeX needs to be run several times, so initially it might get executed on the wrong page. See section 2.8.4 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 \LaTeX has finished the document processing.

2.8.4 Hooks provided \shipout operations

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

2.8.5 Hooks provided by file loading operations

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

2.8.6 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., 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 have hooks in which such support can be added.

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

sffamily Like the `rmfamily` hook but for the `\sffamily` command.

ttfamily Like the `rmfamily` hook but for the `\ttfamily` command.

normalfont The `\normalfont` command resets 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. In the `bfseries/defaults` hook further adjustments can 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 used in `\mdseries` command.

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	H
<code>\<addto-cmd></code> 2	hook commands:
<code>\<env></code> 18	<code>\hook_debug_off:</code> 13
A	<code>\hook_debug_on:</code> 13
<code>\AddToHook</code> 1, 2, 3, 4, 8, 12, 15, 16, 17	<code>\hook_gput_code:nmn</code> 11, 11
<code>\AddToHookNext</code> 4, 4, 8, 12	<code>\hook_gput_next_code:nn</code> 11
<code>\AfterEndEnvironment</code> 18	<code>\hook_gremove_code:nn</code> 12
<code>\AtBeginDocument</code> 2, 6, 16, 17, 17, 19	<code>\hook_gset_rule:nmn</code> 12
<code>\AtBeginDvi</code> 17	<code>\hook_if_empty:nTF</code> 4, 5, 12
<code>\AtBeginEnvironment</code> 18	<code>\hook_if_empty_p:n</code> 12
<code>\AtEndDocument</code> 17, 19, 20	<code>\hook_if_exist:n</code> 9, 12
<code>\AtEndEnvironment</code> 18	<code>\hook_if_exist:nTF</code> 4, 5, 12
<code>\AtEndPreamble</code> 19	<code>\hook_if_exist_p:n</code> 12
B	<code>\hook_log:n</code> 13
<code>\BeforeBeginEnvironment</code> 18, 18, 18, 18	<code>\hook_new:n</code> 11
<code>\begin</code> 1, 18, 19	<code>\hook_new_pair:nn</code> 11
<code>\bfdefault</code> 21	<code>\hook_new_reversed:n</code> 11
<code>\bfseries</code> 21	<code>\hook_show:n</code> 13
C	<code>\hook_use:n</code> 4, 5, 11, 15
<code>\ClearHookRule</code> 8	<code>\hook_use_once:n</code> 4, 11, 15
<code>\clearpage</code> 20	I
<code>\csname</code> 4	<code>\IfHookEmptyTF</code> 4, 5, 8
D	<code>\IfHookExistsTF</code> 4, 5, 9
<code>\DebugHooksOff</code> 10	<code>\ignorespaces</code> 19
<code>\DebugHooksOn</code> 10	<code>\include</code> 21
<code>\DeclareDefaultHookRule</code> 8	<code>\input</code> 6, 21
<code>\DeclareHookRule</code> 1, 3, 7, 8, 10, 12	L
<code>\DeclareHookrule</code> 7	<code>\listfiles</code> 20
<code>\document</code> 19	<code>\LogHook</code> 9
<code>\documentclass</code> 5, 6	M
E	<code>\mdseries</code> 21
<code>\end</code> 1, 18, 19, 21	N
<code>\end(env)</code> 18	<code>\newenvironment</code> 18
<code>\endcsname</code> 4	<code>\NewHook</code> 2, 2, 2, 8, 9, 12, 14, 18
<code>\enddocument</code> 16, 19, 20	<code>\NewMirroredHookPair</code> 2, 9, 12
<code>\errorstopmode</code> 9, 13	<code>\NewReversedHook</code> 2, 2, 6, 9, 12, 14
	<code>\normalfont</code> 21
	<code>\normalsize</code> 3

P		T	
<code>\PopDefaultHookLabel</code>	5, 6, 6	TeX and L ^A T _E X 2 _ε commands:	
<code>\PushDefaultHookLabel</code>	5, 6, 6	<code>\@begindocumenthook</code>	17
R		<code>\@firstofone</code>	2
<code>\RemoveFromHook</code>	3, 3	<code>\@kernel@after@{hookname}</code>	16
<code>\RequirePackage</code>	6	<code>\@kernel@before@{hookname}</code>	16
<code>\rmfamily</code>	21	<code>\@@end</code>	20
S		<code>\expand@font@defaults</code>	21
<code>\selectfont</code>	21	<code>\ttfamily</code>	21
<code>\SetDefaultHookLabel</code>	5, 6, 6	U	
<code>\sffamily</code>	21	<code>\UseHook</code>	2, 4, 5, 13, 15, 16
<code>\shipout</code>	20, 21	<code>\UseOneTimeHook</code>	2, 4, 15
<code>\ShowHook</code>	9, 13, 15	<code>\usepackage</code>	5, 6, 21
<code>\small</code>	3	<code>\usetikzlibrary</code>	6
<code>\special</code>	20		