\LaTeX{}’s hook management*

Frank Mittelbach†

September 23, 2022

Contents

1 Introduction 2

2 Package writer interface 2

2.1 \LaTeX{}2ε interfaces 2

2.1.1 Declaring hooks 2

2.1.2 Special declarations for generic hooks 3

2.1.3 Using hooks in code 3

2.1.4 Updating code for hooks 4

2.1.5 Hook names and default labels 5

2.1.6 The top-level label 7

2.1.7 Defining relations between hook code 8

2.1.8 Querying hooks 9

2.1.9 Displaying hook code 10

2.1.10 Debugging hook code 11

2.2 L3 programming layer (exp13) interfaces 12

2.3 On the order of hook code execution 14

2.4 The use of “reversed” hooks 15

2.5 Difference between “normal” and “one-time” hooks 16

2.6 Generic hooks provided by packages 17

2.7 Private \LaTeX{} kernel hooks 18

2.8 Legacy \LaTeX{}2ε interfaces 18

3 \LaTeX{}2ε commands and environments augmented by hooks 19

3.1 Generic hooks 19

3.1.1 Generic hooks for all environments 20

3.1.2 Generic hooks for commands 21

3.1.3 Generic hooks provided by file loading operations 21

3.2 Hooks provided by \texttt{\begin{document}} 21

3.3 Hooks provided by \texttt{\end{document}} 22

3.4 Hooks provided by \texttt{\shipout} operations 23

3.5 Hooks provided for paragraphs 24

3.6 Hooks provided in NFSS commands 24

3.7 Hook provided by the mark mechanism 25

*This module has version v1.0v dated 2022/06/15, © \LaTeX{} Project.

†Code improvements for speed and other goodies by Phelype Oleinik
1 Introduction

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

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

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

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

2 Package writer interface

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

2.1 \LaTeX{} 2ε interfaces

2.1.1 Declaring hooks

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

\NewHook \NewHook {⟨hook⟩}

Creates a new ⟨hook⟩. If this hook is declared within a package it is suggested that its name is always structured as follows: ⟨package-name⟩/⟨hook-name⟩. If necessary you can further subdivide the name by adding more / parts. If a hook name is already taken, an error is raised and the hook is not created.

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

\NewReversedHook \NewReversedHook {⟨hook⟩}

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

The ⟨hook⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.
A shorthand for \NewHook{⟨hook-1⟩}\NewReversedHook{⟨hook-2⟩}.

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

2.1.2 Special declarations for generic hooks

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

\DisableGenericHook \DisableGenericHook {⟨hook⟩}

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

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

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

\ActivateGenericHook \ActivateGenericHook {⟨hook⟩}

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

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

2.1.3 Using hooks in code

\UseHook \UseHook {⟨hook⟩}

Execute the 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.

\[^1\text{In the 2020/06 release this command was called } \text{\DisableHook, but that name was misleading as it shouldn’t be used to disable non-generic hooks.} \]
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 \addtocmd 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 \addtocmd to simply process its argument, i.e., essentially make it behave like \@firstofone.

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

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

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

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

### 2.1.4 Updating code for hooks

\AddToHook \AddToHook \{\hook\}[\{\label\}]{\{\code\}}

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

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

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

The \hook and \label can be specified using the dot-syntax to denote the current package name. See section 2.1.5.
\RemoveFromHook \RemoveFromHook \{\langle \textit{hook} \rangle \} [\{\langle \textit{label} \rangle \}]

Removes any code labeled by \langle \textit{label} \rangle from the \{\langle \textit{hook} \rangle \}. When the optional argument \langle \textit{label} \rangle is not provided, the \langle \textit{default label} \rangle is used (see section 2.1.5).

If there is no code under the \langle \textit{label} \rangle in the \{\langle \textit{hook} \rangle \}, or if the \{\langle \textit{hook} \rangle \} 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 \langle \textit{label} \rangle 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 \{\langle \textit{hook} \rangle \} and \langle \textit{label} \rangle 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}
\AddToHook{env/quote/before}{\begin{quote}
A quote set in a smaller typeface
\end{quote}}
\RemoveFromHook{env/quote/before}
\AddToHook{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 \{\langle hook\rangle\}\{\langle code\rangle\}

Adds \langle code\rangle to the next invocation of the \langle hook\rangle. 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\{\langle hook\rangle\}\{\langle code-1\rangle\}\AddToHookNext\{\langle hook\rangle\}\{\langle code-2\rangle\}\}

will execute \langle code-1\rangle next time the \langle hook\rangle is used and at that point puts \langle code-2\rangle into the \langle hook\rangle 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 \langle hook\rangle can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\ClearHookNext \ClearHookNext\{\langle hook\rangle\}

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

### 2.1.5 Hook names and default labels

It is best practice to use \AddToHook in packages or classes without specifying a \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

²There is no mechanism to reorder such code chunks (or delete them).
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 \texttt{\csname...\endcsname}, as such, Unicode characters are allowed in \langle hook \rangle and \langle label \rangle arguments). The arguments of \texttt{\UseHook}, \texttt{\UseOneTimeHook}, and \texttt{\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 \texttt{\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 \texttt{mypackage.sty}, the default label is \texttt{mypackage}, so the instructions:

\begin{verbatim}
\NewHook {./hook}
\AddToHook {./hook}[]\{code\}  % Same as \AddToHook{./hook}{code}
\AddToHook {./hook}[]\{sub\}\{code\}
\DeclareHookRule{begindocument}{./}{before}{babel}
\AddToHook {file/foo.tex/after}{code}
\end{verbatim}

are equivalent to:

\begin{verbatim}
\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
\end{verbatim}

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 \texttt{\usepackage} or \texttt{\documentclass}, then the top-level is used as the \langle default label \rangle. This may have exceptions—see \texttt{\PushDefaultHookLabel}.

This syntax is available in all \langle label \rangle arguments and most \langle hook \rangle arguments, both in the \texttt{\LaTeX2} interface, and the \texttt{\LaTeX3} 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 \texttt{\UseHook} (and \texttt{\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 \texttt{\IfHookEmptyTF} (and \texttt{\hook_if_empty:nTF}), because these conditionals are used in some performance-critical parts of the hook management code, and because they are usually used to refer to other package’s hooks, so the dot-syntax doesn’t make much sense.

In some cases, for example in large packages, one may want to separate it in logical parts, but still use the main package name as \langle label \rangle, then the \langle default label \rangle can be set using \texttt{\SetDefaultHookLabel} or \texttt{\PushDefaultHookLabel{...}}\texttt{\PopDefaultHookLabel}. 7
\PushDefaultHookLabel \PushDefaultHookLabel {{default label}}
\PopDefaultHookLabel {code}
\PushDefaultHookLabel
\PushDefaultHookLabel sets the current \textit{default label} to be used in \textit{label} arguments, or when replacing a leading \enquote{.} (see above). \PopDefaultHookLabel reverts the \textit{default label} to its previous value.

Inside a package or class, the \textit{default label} is equal to the package or class name, unless explicitly changed. Everywhere else, the \textit{default label} is \textbf{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{\textit{package name}}{\textit{package code}} \PopDefaultHookLabel to set the \textit{default label} for the package or class file. Inside the \textit{package code} the \textit{default label} can also be changed with \SetDefaultHookLabel. \input and other file input-related commands from the \LaTeX{} kernel do not use \PushDefaultHookLabel, so code within files loaded by these commands does not get a dedicated \textit{label}! (that is, the \textit{default label} is the current active one when the file was loaded.)

Packages that provide their own package-like interfaces (Ti\kZ’s \texttt{usetikzlibrary}, for example) can use \PushDefaultHookLabel and \PopDefaultHookLabel to set dedicated labels and to emulate \usepackage-like hook behavior within those contexts.

The \textbf{top-level} label is treated differently, and is reserved to the user document, so it is not allowed to change the \textit{default label} to \textbf{top-level}.

\SetDefaultHookLabel \SetDefaultHookLabel {{default label}}

Similarly to \PushDefaultHookLabel, sets the current \textit{default label} to be used in \textit{label} arguments, or when replacing a leading \enquote{.}. 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 \textit{default label}.

This command is useful when a large package is composed of several smaller packages, but all should have the same \textit{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 \textit{default label} is \textbf{top-level} and there is no \PopDefaultHookLabel to end its effect. It is also not allowed to change the \textit{default label} to \textbf{top-level}.

\textbf{2.1.6 The top-level label}

The \textbf{top-level} label, assigned to code added from the main document, is different from other labels. Code added to hooks (usually \texttt{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 \textbf{top-level} is always executed at the end of the hook, regardless of where it was declared. If the hook is reversed (see \texttt{NewReversedHook}), the \textbf{top-level} chunk is executed at the very beginning instead.
Rules regarding **top-level** have no effect: if a user wants to have a specific set of rules for a code chunk, they should use a different label to said code chunk, and provide a rule for that label instead. The **top-level** label is exclusive for the user, so trying to add code with that label from a package results in an error.

### 2.1.7 Defining relations between hook code

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

Before the hook management system was introduced packages had to take elaborate precaution to determine of some other package got loaded as well (before or after) and find some ways to alter its behavior accordingly. In addition is was often the user’s responsibility to load packages in the right order so that code added to hooks got added in the right order and some cases even altering the loading order wouldn’t resolve the conflicts.

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

\[\text{\texttt{\textbackslash DeclareHookRule \{\texttt{\textbackslash hook}\}\{\texttt{\textbackslash label1}\}\{\texttt{\textbackslash relation}\}\{\texttt{\textbackslash label2}\}}\]

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

Currently, the supported relations are the following:

- **before** or \(<\): Code for \texttt{\textbackslash label1} comes before code for \texttt{\textbackslash label2}.
- **after** or \(>\): Code for \texttt{\textbackslash label1} comes after code for \texttt{\textbackslash label2}.
- **incompatible-warning** Only code for either \texttt{\textbackslash label1} or \texttt{\textbackslash label2} can appear for that hook (a way to say that two packages—or parts of them—are incompatible). A warning is raised if both labels appear in the same hook.
- **incompatible-error** Like **incompatible-error** but instead of a warning a \LaTeX{} error is raised, and the code for both labels are dropped from that hook until the conflict is resolved.
- **voids** Code for \texttt{\textbackslash label1} overwrites code for \texttt{\textbackslash label2}. More precisely, code for \texttt{\textbackslash 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 \texttt{\textbackslash label1} and \texttt{\textbackslash 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 \texttt{\textbackslash DeclareHookRule} overwrites any previous declaration.

The \texttt{\textbackslash hook} and \texttt{\textbackslash label} can be specified using the dot-syntax to denote the current package name. See section 2.1.5.
Syntactic sugar for saying that ⟨label1⟩ and ⟨label2⟩ are unrelated for the given ⟨hook⟩.

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

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

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

2.1.8 Querying hooks

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

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

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

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

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

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

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

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

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

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

\ShowHook \ShowHook \LogHook \LogHook

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 \texttt{TeX}'s prompt (only in \texttt{errorstopmode}) to wait for user action.

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

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

```
-> The hook 'example-hook':
> Code chunks:
>   foo -> [code from package 'foo']
>   bar -> [from package 'bar']
>   baz -> [package 'baz' is here]
> Document-level (top-level) code (executed last):
>   -> [code from 'top-level']
> Extra code for next invocation:
>   -> [one-time code]
> Rules:
>   foo|baz with relation >
>   baz|bar with default relation <
> Execution order (after applying rules):
>   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

\texttt{(label) \rightarrow (code)}

Line 7 shows the code chunk added by the user in the main document (labeled \texttt{top-level}) in the format

\texttt{Document-level (top-level) code (executed (first|last))}:

\texttt{-> (top-level code)}
This code will be either the first or last code executed by the hook (\texttt{last} if the hook is normal, \texttt{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

\[-\rangle \langle \text{next-code} \rangle\]

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

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

\(\langle label-1 \rangle | \langle label-2 \rangle \text{ with (default?)} \text{ relation (relation)}\)

which means that the (relation) applies to \(\langle label-1 \rangle\) and \(\langle label-2 \rangle\), in that order, as detailed in \texttt{\DeclareHookRule}. If the relation is \texttt{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

\texttt{\DebugHooksOn} \texttt{\DebugHooksOff}

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

### 2.2 L3 programming layer (\texttt{expl3}) interfaces

This is a quick summary of the \texttt{L\LaTeX3} programming interfaces for use with packages written in \texttt{expl3}. In contrast to the \texttt{\LaTeX2\epsilon} 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 \texttt{expl3} packages, but the choice is yours.

\begin{verbatim}
\hook_new:n \hook_new_reversed:n \hook_new_pair:nn
\end{verbatim}

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

Creates a new \(\langle hook \rangle\) with normal or reverse ordering of code chunks. \texttt{\hook_new_-pair:nn} creates a pair of such hooks with \{\texttt{\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.

\begin{verbatim}
\hook_disable_generic:n
\end{verbatim}

\hook_disable_generic:n \{\langle hook \rangle\}

Marks \{\texttt{\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 \texttt{\hook_use:n} will simply do nothing.

This declaration is intended for use with generic hooks that are known not to work (see \texttt{\LaTeXcmdhooks-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.
\hook_activate_generic:n \hook_activate_generic:n {\hook}

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

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

\hook_use:n \hook_use:n {\hook}

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

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

\hook_use_once:n \hook_use_once:n {\hook}

Changes the \hook status so that from now on any addition to the hook code is executed immediately. Then execute any \hook code already set up.

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

\hook_gput_code:nnn \hook_gput_code:nnn {\hook} \{\label\} \{\code\}

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

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

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

\hook_gput_next_code:nn \hook_gput_next_code:nn {\hook} \{\code\}

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

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

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

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

\hook_gclear_next_code:n \hook_gclear_next_code:n {\hook}

Undo any earlier \hook_gput_next_code:nn.
\hook_gremove_code:nn  \hook_gremove_code:nn \{\hook\} \{\{\label\}\}\{\}\n
Removes any code for \{\hook\} labeled \{\label\}.

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

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

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

\hook_gset_rule:nn  \hook_gset_rule:nn \{\hook\} \{\{\label1\}\} \{\{\label2\}\} \{\{\relation\}\} \{\}\n
Relate \{\label1\} with \{\label2\} when used in \{\hook\}. See \DeclareHookRule for the allowed \{\relation\}s. If \{\hook\} is ?? a default rule is specified.

The \{\hook\} and \{\label\} can be specified using the dot-syntax to denote the current package name. See section 2.1.5. The dot-syntax is parsed in both \{\label\} arguments, but it usually makes sense to be used in only one of them.

\hook_if_empty_p:n  \hook_if_empty:nTF \{\hook\} \{\{\true\}\} \{\{\false\}\} \{\}\n
Tests if the \{\hook\} is empty (i.e., no code was added to it using either \AddToHook or \AddToHookNext), and branches to either \{\true\} or \{\false\} depending on the result.

The \{\hook\} cannot be specified using the dot-syntax. A leading . is treated literally.

\hook_show:n  \hook_log:n \{\hook\} \{\}\n
Displays information about the \{\hook\} such as

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

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

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

\hook_debug_on:  \hook_debug_off: \{\}\n
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
\DeclarationsHookRule{myhook}{packageB}{before}{packageC}
the execution order would not have changed.

In summary: it is not possible to rely on the order of execution unless there are rules
that partially or fully define the order (in which you can rely on them being fulfilled).

2.4 The use of “reversed” hooks

You may have wondered why you can declare a “reversed” hook with \NewReversedHook
and what that does exactly.

In short: the execution order of a reversed hook (without any rules!) is exactly
reversed to the order you would have gotten for a hook declared with \NewHook.

This is helpful if you have a pair of hooks where you expect to see code added that
involves grouping, e.g., starting an environment in the first and closing that environment
in the second hook. To give a somewhat contrived example\(^4\), suppose there is a package
adding the following:

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

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

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

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

package-1, package-too

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

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

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

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

\(^4\)There are simpler ways to achieve the same effect.
> ---
> Extra code for next invocation:
> ---
> Rules:
> ---
> Execution order (after reversal):
>   package-too, package-1.

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

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

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

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

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

This has some consequences one needs to be aware of:

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

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

In particular this means that construct such as

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

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

- execute ⟨code-1⟩,

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

- and finally execute ⟨code-3⟩.

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

\(^5\)This is sometimes used with `\AtBeginDocument` which is why it is supported.
2.6 Generic hooks provided by packages

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

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

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

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

For example, babel may want to provide hooks such as babel/⟨language⟩/afterextras. 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 considerabily.

2.7 Private \LaTeX kernel hooks

There are a few places where it is absolutely essential for \LaTeX to function correctly that code is executed in a precisely defined order. Even that could have been implemented with the hook management (by adding various rules to ensure the appropriate ordering with respect to other code added by packages). However, this makes every document unnecessary slow, because there has to be sorting even 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@\langle hook\rangle or \@kernel@after@\langle hook\rangle. For example, in \texttt{\enddocument} you find

\texttt{\UseHook{enddocument} \@kernel@after@enddocument}

which means first the user/package-accessible \texttt{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.\footnote{As with everything in \TeX{} there is not enforcement of this rule, and by looking at the code it is easy to find out how the kernel adds to them. The main reason of this section is therefore to say “please don’t do that, this is unconfigurable code!”}

\section*{2.8 Legacy \LaTeX{} 2ε interfaces}

\LaTeX{} 2ε offered a small number of hooks together with commands to add to them. They are listed here and are retained for backwards compatibility.

With the new hook management, several additional hooks have been added to \LaTeX{} and more will follow. See the next section for what is already available.

\begin{verbatim}
\AtBeginDocument \AtBeginDocument \langle label\rangle \{ \langle code\rangle \}
\end{verbatim}

If used without the optional argument \texttt{\langle label\rangle}, it works essentially like before, i.e., it is adding \texttt{\langle code\rangle} to the hook \texttt{begindocument} (which is executed inside \texttt{\begin{document}}). However, all code added this way is labeled with the label \texttt{top-level} (see section 2.1.6) if done outside of a package or class or with the package/class name if called inside such a file (see section 2.1.5).

This way one can add further code to the hook using \texttt{\AddToHook} or \texttt{\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 \texttt{\AddToHook \langle begindocument\rangle \langle label\rangle \{ \langle code\rangle \}}.

\texttt{\AtBeginDocument} is a wrapper around the \texttt{begindocument} hook (see section 3.2), which is a one-time hook. As such, after the \texttt{begindocument} hook is executed at \texttt{\begin{document}} any attempt to add \texttt{\langle code\rangle} to this hook with \texttt{\AtBeginDocument} or with \texttt{\AddToHook} will cause that \texttt{\langle code\rangle} 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.

\begin{verbatim}
\AtEndDocument \AtEndDocument \langle label\rangle \{ \langle code\rangle \}
\end{verbatim}

Like \texttt{\AtBeginDocument} but for the \texttt{enddocument} hook.

The few hooks that existed previously in \LaTeX{} 2ε used internally commands such as \texttt{\@begindocumenthook} and packages sometimes augmented them directly rather than working through \texttt{\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 \texttt{legacy} so that it doesn’t get lost.

However, over time the remaining cases of direct usage need updating because in one of the future release of \LaTeX{} we will turn this legacy support off, as it does unnecessary slow down the processing.
3 \LaTeX{} 2ε commands and environments augmented by hooks

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

3.1 Generic hooks

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

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

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

- \texttt{env/⟨env⟩/before} This hook is executed as part of \texttt{\begin} as the very first action, in particular prior to starting the environment group. Its scope is therefore not restricted by the environment.
- \texttt{env/⟨env⟩/begin} This hook is executed as part of \texttt{\begin} directly in front of the code specific to the environment start (e.g., the second argument of \texttt{\newenvironment}). Its scope is the environment body.
- \texttt{env/⟨env⟩/end} This hook is executed as part of \texttt{\end} directly in front of the code specific to the end of the environment (e.g., the third argument of \texttt{\newenvironment}).
env/{env}/after This hook is executed as part of \end after the code specific to the environment end and after the environment group has ended. Its scope is therefore not restricted by the environment.

The hook is implemented as a reversed hook so if two packages add code to env/{env}/before and to env/{env}/after they can add surrounding environments and the order of closing them happens in the right sequence.

Generic environment hooks are never one-time hooks even with environments that are supposed to appear only once in a document.\(^7\) In contrast to other hooks there is also no need to declare them using \NewHook. The hooks are only executed if \begin{env} and \end{env} is used. If the environment code is executed via low-level calls to \begin{env} and \end{env} (e.g., to avoid the environment grouping) they are not available. If you want them available in code using this method, you would need to add them yourself, i.e., write something like

\begin{quote}
\end{quote}\UseHook{env/quote/before}\quote...
\endquote\UseHook{env/quote/after}

to add the outer hooks, etc.

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

\BeforeBeginEnvironment \BeforeBeginEnvironment \[\begin{label}\{\text{env}\}\end{label}\}{\{\text{code}\}}

This declaration adds to the env/{env}/before hook using the \begin{label}. If \begin{label} is not given, the \begin{default label} is used (see section 2.1.5).

\AtBeginEnvironment \AtBeginEnvironment \[\begin{label}\{\text{env}\}\end{label}\}{\{\text{code}\}}

This is like \BeforeBeginEnvironment but it adds to the env/{env}/begin hook.

\AtEndEnvironment \AtEndEnvironment \[\begin{label}\{\text{env}\}\end{label}\}{\{\text{code}\}}

This is like \BeforeBeginEnvironment but it adds to the env/{env}/end hook.

\AfterEndEnvironment \AfterEndEnvironment \[\begin{label}\{\text{env}\}\end{label}\}{\{\text{code}\}}

This is like \BeforeBeginEnvironment but it adds to the env/{env}/after hook.

3.1.2 Generic hooks for commands

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

cmd/{name}/before This hook is executed at the very start of the command execution.

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

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

\(^7\)Thus if one adds code to such hooks after the environment has been processed, it will only be executed if the environment appears again and if that doesn’t happen the code will never get executed.
3.1.3 Generic hooks provided by file loading operations

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

3.2 Hooks provided by \texttt{\begin{document}}

Until 2020 \texttt{\begin{document}} offered exactly one hook that one could add to using \texttt{\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 \texttt{etoolbox} and others that augmented \texttt{document} to gain better control.

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

\texttt{begindocument/before} This hook is executed at the very start of \texttt{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 \texttt{etoolbox}'s \texttt{\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).

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

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

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

\texttt{begindocument/end} This hook is executed at the end of the \texttt{document} code in other words at the beginning of the document body. The only command that follows it is \texttt{\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 \texttt{\begin} also exist, i.e., \texttt{env/document/before} and \texttt{env/document/end}, but with this special environment it is better use the dedicated one-time hooks above.

3.3 Hooks provided by \texttt{\end{document}}

\LaTeX{} always provided \texttt{\AtEndDocument} to add code to the execution of \texttt{\end{document}} just in front of the code that is normally executed there. While this was a big improvement over the situation in \LaTeX{} 2.09 it was not flexible enough for a number of use cases and so packages, such as \texttt{etoolbox}, \texttt{atveryend} and others patched \texttt{\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 lengthy) list of files if that got requested via \listfiles.

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

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

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

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

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

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 2.5). Is it even possible to add code after this one?
There is also the hook \shipout/lastpage. This hook is executed as part of the last \shipout in the document to allow package to add final \special's to that page. Where this hook is executed in relation to those from the above list can vary from document to document. Furthermore to determine correctly which of the \shipouts is the last one, \LaTeX needs to be run several times, so initially it might get executed on the wrong page. See section 3.4 for where to find the details.

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

3.4 Hooks provided by \shipout operations

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

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

\texttt{sffamily} This is like the \texttt{rmfamily} hook, but for the \texttt{sffamily} command.

\texttt{ttfamily} This is like the \texttt{rmfamily} hook, but for the \texttt{ttfamily} command.

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

\texttt{expand@font@defaults} The internal \texttt{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.

\texttt{bfseries/defaults}, \texttt{bfseries} If the \texttt{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 \texttt{bfseries} is called. The \texttt{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 \texttt{bfseries} hook is always executed just before \texttt{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.

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                                    | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|--------------------------------------------|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|}

<table>
<thead>
<tr>
<th>A</th>
<th>\ActivateGenericHook</th>
<th>\AddToHook</th>
<th>\AddToHookNext</th>
<th>\AfterEndEnvironment</th>
<th>\AtBeginDocument</th>
<th>\AtBeginEnvironment</th>
<th>\AtEndDocument</th>
<th>\AtEndEnvironment</th>
<th>\AtEndPreamble</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>\BeforeBeginEnvironment</td>
<td>\begin</td>
<td>\bfdefault</td>
<td>\bfseries</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>\ClearHookNext</td>
<td>\ClearHookRule</td>
<td>\clearpage</td>
<td>\csname</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D \DebugHooksOff
\DebugHooksOn
\DeclareDefaultHookRule
\DeclareHookRule
\DisableGenericHook
\DisableHook
\DiscardShipoutBox
\document
\documentclass
\end
\endcsname
\enddocument
\errorstopmode
\errorstopmode

E

H hook commands:
\hook_activate_generic:n
\hook_debug_off:
\hook_debug_on:
\hook_disable_generic:n
\hook_gclear_next_code:n
\hook_gput_code:nnn

25