The \texttt{lthooks} package\footnote{This package has version v1.0a dated 2020/09/10, © \LaTeX{} Project.}

Frank Mittelbach\footnote{Code improvements for speed and other goodies by Phelype Oleinik

September 11, 2020

Contents

1 Introduction \hfill 2

2 Package writer interface \hfill 2
  2.1 \LaTeX{}2\e interfaces \hfill 2
     \hspace{0.5cm} 2.1.1 Declaring hooks and using them in code \hfill 2
     \hspace{0.5cm} 2.1.2 Updating code for hooks \hfill 3
     \hspace{0.5cm} 2.1.3 Hook names and default labels \hfill 5
     \hspace{0.5cm} 2.1.4 Defining relations between hook code \hfill 6
     \hspace{0.5cm} 2.1.5 Querying hooks \hfill 7
     \hspace{0.5cm} 2.1.6 Displaying hook code \hfill 8
     \hspace{0.5cm} 2.1.7 Debugging hook code \hfill 9
     \hspace{0.5cm} 2.2 L3 programming layer (exp3) interfaces \hfill 9
     \hspace{0.5cm} 2.3 On the order of hook code execution \hfill 11
     \hspace{0.5cm} 2.4 The use of “reversed” hooks \hfill 13
     \hspace{0.5cm} 2.5 Difference between “normal” and “one-time” hooks \hfill 14
     \hspace{0.5cm} 2.6 Private \LaTeX{} kernel hooks \hfill 14
     \hspace{0.5cm} 2.7 Legacy \LaTeX{}2\e interfaces \hfill 15
     \hspace{0.5cm} 2.8 \LaTeX{}2\e commands and environments augmented by hooks \hfill 15
        \hspace{0.5cm} 2.8.1 Generic hooks for all environments \hfill 16
        \hspace{0.5cm} 2.8.2 Hooks provided by \texttt{\begin{document}} \hfill 17
        \hspace{0.5cm} 2.8.3 Hooks provided by \texttt{\end{document}} \hfill 17
        \hspace{0.5cm} 2.8.4 Hooks provided \texttt{\shipout} operations \hfill 18
        \hspace{0.5cm} 2.8.5 Hooks provided by file loading operations \hfill 18
        \hspace{0.5cm} 2.8.6 Hooks provided in NFSS commands \hfill 19

Index \hfill 19
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\ 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.

\begin{verbatim}
\NewHook \NewHook \{\hook\}
\end{verbatim}

Creates a new \langle\hook\rangle. If this is a hook provided as part of a package it is suggested that the \langle\hook\rangle name is always structured as follows: \langlepackage-name\rangle/\langlehook-name\rangle. 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 \langle\hook\rangle can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

\begin{verbatim}
\NewReversedHook \NewReversedHook \{\hook\}
\end{verbatim}

Like \NewHook declares a new \langle\hook\rangle, 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 \langle\hook\rangle can be specified using the dot-syntax to denote the current package name. See section 2.1.3.
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{⟨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{⟨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 \AddToHook 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.

*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{⟨hook⟩}{⟨label⟩}{⟨code⟩}

Adds ⟨code⟩ to the ⟨hook⟩ labeled by ⟨label⟩. If the optional argument ⟨label⟩ is not provided, if \AddToHook is used in a package/class, then the current package/class name is used, otherwise top-level is used (see section 2.1.3).

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 later 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 hook of other packages without worrying whether they are actually used in the current document. See section 2.1.5.

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

Removes any code labeled by \langle label \rangle from the \langle hook \rangle. If the optional argument \langle label \rangle is not provided, if \AddToHook is used in a package/class, then the current package/class name is used, otherwise top-level is used.

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 \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.

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 \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 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.\footnote{There is no mechanism to reorder such code chunks (or delete them).}

It is possible to nest declarations 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 2.1.5.

The \langle hook \rangle 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 and \UseOneTimeHook 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 \langle hook \rangle name or \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 \DeclareDefaultHookLabel). 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 to the name of the current package or class (using \@currname). If \@currname is not set (because the hook command is used outside of a package, or the current file wasn't loaded with \usepackage or \documentclass), then the top-level is used as the \langle default label \rangle.

This syntax is available in all \langle label \rangle arguments and most \langle hook \rangle, both in the \LaTeX 2ε interface, and the \LaTeX 3 interface described in section 2.2.
Note, however, that the replacement of \( \langle \text{default label} \rangle \) by the \( \langle \text{default label} \rangle \) takes place when the hook command is executed, so actions that are somehow executed after the package ends will have the wrong \( \langle \text{default label} \rangle \) if the dot-syntax is used. For that reason, this syntax is not available in \texttt{\usehook} (and \texttt{\hookuse: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{\hookifempty:nTF}) and \texttt{\ifhookexistsTF} (and \texttt{\hookifexist: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 \( \langle \text{label} \rangle \), then the \( \langle \text{default label} \rangle \) can be set using \texttt{\declaredefaulthooklabel}:

\begin{verbatim}
\DeclareDefaultHookLabel \{ \langle \text{default label} \rangle \}
\end{verbatim}

Sets the \( \langle \text{default label} \rangle \) to be used in \( \langle \text{label} \rangle \) arguments. If \texttt{\DeclareDefaultHookLabel} is not used in the current package, \texttt{@currname} is used instead. If \texttt{@currname} is not set, the code is assumed to be in the main document, in which case \texttt{top-level} is used.

The effect of \texttt{\DeclareDefaultHookLabel} holds for the current file, and is reset to the previous value when the file is closed.

### 2.1.4 Defining relations between hook code

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

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

With the new hook management system it is now possible to define rules (i.e., relationships) between code chunks added by different packages and explicitly describe in which order they should be processed.
\DeclareHookRule{⟨hook⟩}{⟨label1⟩}{⟨relation⟩}{⟨label2⟩}

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

Currently, the supported relations are the following:

- **before** or **<** Code for ⟨label1⟩ comes before code for ⟨label2⟩.
- **after** or **>** Code for ⟨label1⟩ comes after code for ⟨label2⟩.

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

**incompatible-error** Like **incompatible-error** but instead of a warning a \LaTeX error is raised, and the code for both labels are dropped from that hook until the conflict is resolved.

**voids** Code for ⟨label1⟩ overwrites code for ⟨label2⟩. More precisely, code for ⟨label2⟩ is dropped for that hook. This can be used, for example if one package is a superset in functionality of another one and therefore wants to undo code in some hook and replace it with its own version.

**unrelated** The order of code for ⟨label1⟩ and ⟨label2⟩ is irrelevant. This rule is there to undo an incorrect rule specified earlier.

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

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

\ClearHookRule{⟨hook⟩}{⟨label1⟩}{⟨label2⟩}

Syntactic sugar for saying that ⟨label1⟩ and ⟨label2⟩ are unrelated for the given ⟨hook⟩.

\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.

### 2.1.5 Querying hooks

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

- exist and be empty;

\(^2\)Trying 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 \texttt{A} defines hook \texttt{A/foo}, and package \texttt{B} adds some code to that hook. However, a document may load package \texttt{B} before package \texttt{A}, or may not load package \texttt{A} at all. In both cases some code is added to hook \texttt{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 \texttt{\texttt{\textbackslash NewHook}} or some variant thereof.

\begin{verbatim}
\IfHookEmptyTF \{\texttt{\textbackslash hook}\} \{\texttt{true code}\} \{\texttt{false code}\}
\end{verbatim}

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

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

\begin{verbatim}
\IfHookExistsTF \{\texttt{\textbackslash hook}\} \{\texttt{true code}\} \{\texttt{false code}\}
\end{verbatim}

Tests if the \texttt{\textbackslash hook} exists (if it was created with either \texttt{\textbackslash NewHook}, \texttt{\textbackslash NewReversedHook}, or \texttt{\textbackslash NewMirroredHookPair}), and branches to either \texttt{true code} or \texttt{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 \texttt{\textbackslash hook\_if\_exist:n} will change once code is added to said hook (unless the hook was previously declared).

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

\texttt{FMi: Would be helpful if we provide some use cases}

\subsection*{2.1.6 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

Displays information about the (hook) such as

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

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

The (hook) can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

2.1.7 Debugging hook code

\DebugHooksOn \DebugHooksOff

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 (exp3) interfaces

This is a quick summary of the \LaTeX3 programming interfaces for use with packages written in \exp3. In contrast to the \LaTeX2ε 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 \exp3 packages, but the choice is yours.

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

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:nn

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 \langle \text{hook} \rangle 

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

\textit{FMi: better L3 name?}

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

\hook_gput_code:nnn \hook_gput_code:nnn \langle \text{hook} \rangle \{\langle \text{label} \rangle \} \{\langle \text{code} \rangle \}

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

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

The \langle \text{hook} \rangle and \langle \text{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 \text{hook} \rangle \{\langle \text{code} \rangle \}

Adds a chunk of \langle \text{code} \rangle for use only in the next invocation of the \langle \text{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 \text{code} \rangle.

The \langle \text{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 \text{hook} \rangle \{\langle \text{label} \rangle \}

Removes any code for \langle \text{hook} \rangle labeled \langle \text{label} \rangle.

If the code for that \langle \text{label} \rangle wasn’t yet added to the \langle \text{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 \text{hook} \rangle and \langle \text{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 \text{hook} \rangle \{\langle \text{label1} \rangle \} \{\langle \text{relation} \rangle \} \{\langle \text{label2} \rangle \}

Relate \langle \text{label1} \rangle with \langle \text{label2} \rangle when used in \langle \text{hook} \rangle. See \texttt{\textbackslash DeclareHookRule} for the allowed \langle \text{relation} \rangle s. If \langle \text{hook} \rangle is ?? a default rule is specified.

The \langle \text{hook} \rangle and \langle \text{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 \text{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 \{\hook\} \{\{true \; code\}\} \{\{false \; code\}\}

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 \star \hook_if_exist:nTF \{\hook\} \{\{true \; code\}\} \{\{false \; code\}\}

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 \star \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: \star \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}
Extra code 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, or 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}

then we get

The hook 'myhook':
Code chunks:
  packageA -> \typeout {A}
  packageB -> \typeout {B}
  packageC -> \typeout {C}
Extra code next invocation:
  ---
Rules:
  packageA|packageB with relation before
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 \texttt{\texttt{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 \texttt{\texttt{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\(^3\), suppose there is a package adding the following:

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

As a result, all quotes will be in italics. Now suppose further that the user wants the quotes also in blue and therefore adds:

\begin{verbatim}
\usepackage{color}
\AddToHook{env/quote/before}{\begin{color}{blue}}
\AddToHook{env/quote/after} {\end{color}}
\end{verbatim}

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

\begin{verbatim}
package-1, top-level
\end{verbatim}

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

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

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

The hook ‘env/quote/after’:

Code chunks:
- package-1 -> \end{itshape}
- top-level -> \end{color}

Extra code next invocation:

---

Rules:

---

Execution order (after reversal):
- top-level, 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.

\(^3\)there 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 \textit{⟨code⟩} immediately.

This has some consequences one needs to be aware of:

- If \textit{⟨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 \textit{⟨code⟩} will never be executed.

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

In particular this means that construct such as

\begin{Verbatim}
\AddToHook{myhook}
  \{ \textit{⟨code-1⟩} \AddToHook{myhook}\{\textit{⟨code-2⟩}\} \textit{⟨code-3⟩} \}
\end{Verbatim}

works for one-time hooks\footnote{This is sometimes used with `\AtBeginDocument` which is why it is supported.} (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 `\AddToHook{myhook}` is executed it would

- execute \textit{⟨code-1⟩},

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

- and finally execute \textit{⟨code-3⟩}.

The second time `\UseHook` is called it would execute the above and in addition \textit{⟨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 \textit{⟨code-2⟩} is added and so that code chunk is executed \textit{(⟨# of invocations⟩ − 1)} times.

2.6 Private \TeX{} kernel hooks

There are a few places where it is absolutely essential for \TeX{} 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 unnecessarily 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 \TeX{}).

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
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{Legacy \LaTeX{}\textcopyright{} interfaces}

\LaTeX{}\textcopyright{} 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 \{⟨label⟩\} \{⟨code⟩\}
\end{verbatim}

If used without the optional argument \texttt{⟨label⟩}, it works essentially like before, i.e., it is adding \texttt{⟨code⟩} 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} if done outside of a package or class or with the package/class name if called inside such a file.

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 the \texttt{top-level} code. When using the optional argument the call is equivalent to running \texttt{\AddToHook \{begindocument\} \{⟨label⟩\} \{⟨code⟩\}}.

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 \{⟨label⟩\} \{⟨code⟩\}
\end{verbatim}

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

\begin{verbatim}
\AtBeginDvi \{⟨label⟩\} \{⟨code⟩\}
\end{verbatim}

This hook is discussed in conjunction with the shipout hooks.

The few hooks that existed previously in \LaTeX{}\textcopyright{} 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.

\section{\LaTeX{}\textcopyright{} commands and environments augmented by hooks}

\textit{intro to be written}
2.8.1 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}).

\texttt{env/env/after} \ This hook is executed as part of \texttt{\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 \texttt{env/env/before} and to \texttt{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.\textsuperscript{6} In contrast to other hooks there is also no need to declare them using \texttt{\NewHook}.

The hooks are only executed if \texttt{\begin\{env\}} and \texttt{\end\{env\}} is used. If the environment code is executed via low-level calls to \texttt{\begin} and \texttt{\end} (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{verbatim}
\UseHook{env/quote/before}\quote
...
\endquote\UseHook{env/quote/after}
\end{verbatim}

to add the outer hooks, etc.

\begin{verbatim}
\BeforeBeginEnvironment \BeforeBeginEnvironment [\{label\}] \{\{code\}\}
This declaration adds to the env\{env\}/before hook using by default the current package or class name as a label or top-level if used in the document directly.

\AtBeginEnvironment \AtBeginEnvironment [\{label\}] \{\{code\}\}
Like \BeforeBeginEnvironment but adds to the env\{env\}/begin hook.

\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.
\end{verbatim}

\textsuperscript{6}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.
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):

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

- **begindocument** 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 `\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.

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

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}`

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

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

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

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

  When this hook is executed there may be still unprocessed material (e.g., floats on the deferlist) and the hook may add further material to be typeset. After it, `\clearpage` is called to ensure that all such material gets typeset. If there is nothing waiting the `\clearpage` has no effect.
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 \texttt{.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 \texttt{.aux} file is re-read in the next step.

After this hook has been executed the \texttt{.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.).

enddocument/afteraux At this point, the \texttt{.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 \texttt{\listfiles}.

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 \texttt{kernel/filelist} and \texttt{kernel/warnings}), namely the list of files when \texttt{\listfiles} has been used and the warnings for duplicate labels, missing references, font substitutions etc.

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

There is also the hook \texttt{shipout/lastpage}. This hook is executed as part of the last \texttt{\shipout} in the document to allow package to add final \texttt{\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 \texttt{\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 \texttt{env/document/end} hook which is executed by \texttt{\end}, i.e., just in front of the first hook above. Note however that the other generic \texttt{\end} environment hook, i.e., \texttt{env/document/after} will never get executed, because by that time \LaTeX has finished the document processing.

2.8.4 Hooks provided \texttt{\shipout} operations

There are several hooks and mechanisms added to \LaTeX’s process of generating pages. These are documented in \texttt{ltshipout-doc.pdf} or with code in \texttt{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 \texttt{\input}, \texttt{\include}, \texttt{\usepackage}, etc. These are documented in \texttt{ltfilehook-doc.pdf} or with code in \texttt{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.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>\AtEndDocument ............... 14, 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{addto-cmd})</td>
<td>\AtEndEnvironment .................. 15</td>
</tr>
<tr>
<td>(\text{env})</td>
<td>\AtEndPreamble ...................... 16</td>
</tr>
<tr>
<td>(\text{goals})</td>
<td>\end ................................ 15, 15, 15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>\AddToHook ............. 1, 2, 3, 4, 7, 10, 13, 14</td>
<td>\BeforeBeginEnvironment .... 15, 15, 15, 15</td>
</tr>
<tr>
<td>\AddToHookNext .......... 3, 4, 7, 10</td>
<td>\begin ................................ 1, 15, 16</td>
</tr>
<tr>
<td>\AfterEndEnvironment    15</td>
<td>\bfdefault .......................... 18</td>
</tr>
<tr>
<td>\AtBeginDocument         14</td>
<td>\bfseries ........................... 18</td>
</tr>
<tr>
<td>\AtBeginDocument         2, 15, 14, 16</td>
<td>\AtBeginEnv ................................ 14</td>
</tr>
<tr>
<td>\AtBeginDocui            15</td>
<td>\ClearHookRule ........................ 6</td>
</tr>
<tr>
<td>\AtBeginEnvironment     15</td>
<td>\ClearHookRule ........................ 6</td>
</tr>
</tbody>
</table>

19