LaTeX's socket management*

Frank Mittelbach

November 25, 2024

Abstract

This code implements sockets which are places in the code into which predeclared chunks of code (plugs) can be placed. Both the sockets and the plugs are "named" and each socket is assigned exactly one plug at any given time.

Contents

1 Introduction **1**

2 Configuration of the transformation process **2**
2.1 The template mechanism 2
2.2 The hook mechanism 2
2.3 The socket mechanism 3
2.3.1 Examples 4
2.3.2 Details and semantics 6
2.3.3 Command syntax 7
2.3.4 Rationale for error handling 9

Index **9**

1 Introduction

A LaTeX source file is transformed into a typeset document by executing code for each command or environment in the document source. Through various steps this code transforms the input and eventually generates typeset output appearing in a "galley" from which individual pages are cut off in an asynchronous way. This page generating process is normally not directly associated with commands in the input[1] but is triggered whenever the galley has received enough material to form another page (giving current settings).

As part of this transformation input data may get stored in some form and later reused, for example, as part of the output routine processing.

*This module has version v0.9b dated 2024/10/27, © LaTeX Project.

[1] Excepts for directives such as `\newpage`.

2 Configuration of the transformation process

There are three different major methods offered by LaTeX to configure the transformation process:

- through the template mechanism,
- through the hook mechanism, or
- through sockets and plugs.

They offer different possibilities (with different features and limitations) and are intended for specific use cases, though it is possible to combine them.

2.1 The template mechanism

The template mechanism is intended for more complex document-level elements (e.g., headings such as `\section` or environments like `itemize`). The template code implements the overall processing logic for such an element and offers a set of parameters to influence the final result.

The document element is then implemented by (a) selecting a suitable template (there may be more than one available for the kind of document element) and (b) by setting its parameters to desired values. This then forms a so-called instance which is executed when the document element is found in the source.

By altering the parameter values (in a document class or in the document preamble) or, if more drastic layout changes are desired, by selecting a different template and then adjusting its parameters, a wide variety of layouts can be realized through simple configuration setups without the need to develop new code.

The target audience of this method are therefore document class developers or users who wish to alter an existing layout (implemented by a document class) in certain (minor) ways.

The template mechanism is currently documented as part of the `xtemplate` package and one more elaborate implementation can be found as part of the `latex-lab` code for lists (to be documented further).

2.2 The hook mechanism

Hooks are places in the kernel code (or in packages) that offer packages the possibility to inject additional code at specific points in the processing in a controlled way without the need to replace the existing code block (and thereby overwriting modifications/extensions made by other packages). The target audience is therefore mainly package developers, even though some hooks can be useful for document authors.

Obviously, what can reasonably be added into a hook depends on the individual hook (hopefully documented as part of the hook documentation), but in general the idea behind hooks is that more than one package could add code into the hook at the same time. Perhaps the most famous hook (that LaTeX had for a very long time) is `begindocument` into which many packages add code to through `\AtBeginDocument{`⟨*code*⟩`}` (which is nowadays implemented as a shorthand for `\AddToHook{begindocument}{`⟨*code*⟩`}`). To resolve possible conflicts between injections by different packages there is a rule mechanism by which code chunks in a hook can be ordered in a certain way and by which incompatible packages can be detected if a resolution is impossible.

In contrast to template code, there is no standard configuration method through parameters for hooks, i.e., the code added to a hook "is" the configuration. If it wants to provide for configuration through parameters it has to also provide its own method to set such parameters in some way. However, in that case it is likely that using a hook is not the right approach and the developer better calls a template instance instead which then offers configuration through a key/value interface.

In most cases, hooks do not take any arguments as input. Instead, the data that they can (and are allowed to) access depends on the surrounding context.

For example, the various hooks available during the page shipout process in LaTeX's output routine can (and have to) access the accumulated page material stored in a box named `\ShipoutBox`. This way, code added to, say, the `shipout/before` hook could access the page content, alter it, and then write it back into `\ShipoutBox` and any other code added to this hook could then operate on the modified content. Of course, for such a scheme to work the code prior to executing the hook would need to setup up data in appropriate places and the hook documentation would need to document what kind of storage can be accessed (and possibly altered) by the hook.

There are also hooks that take arguments (typically portions of document data) and in that case the hook code can access these arguments through `#1`, `#2`, etc.

The hook mechanism is documented in `lthooks-doc.pdf`.

2.3 The socket mechanism

In some cases there is code that implements a certain programming logic (for example, combining footnotes, floats, and the text for the current page to be shipped out) and if this logic should change (e.g., footnotes to be placed above bottom floats instead of below) then this whole code block needs to be replaced with different code.

In theory, this could be implemented with templates, i.e., the code simply calls some instance that implements the logic and that instance is altered by selecting a different templates and/or adjusting their parameters. However, in many cases customization through parameters is overkill in such a case (or otherwise awkward, because parameterization is better done on a higher level instead of individually for small blocks of code) and using the template mechanism just to replace one block of code with a different one results in a fairly high performance hit. It is therefore usually not a good choice.

In theory, it would also be possible to use a hook, but again that is basically a misuse of the concept, because in this use case there should never be more than one block of code inside the hook; thus, to alter the processing logic one would need to set up rules that replace code rather than (as intended) execute all code added to the hook.

For this reason LaTeX now offers a third mechanism: "sockets" into which one can place exactly one code block — a "plug".

In a nutshell: instead of having a fixed code block somewhere as part of the code, implementing a certain programming logic there is a reference to a named socket at this point. This is done by first declaring the named socket with:

`\NewSocket{`⟨*socket-name*⟩`}{`⟨*number-of-inputs*⟩`}`

This is then referenced at the point where the replaceable code block should be executed with:

`\UseSocket{`⟨*socket-name*⟩`}`

or, if the socket should take a number of inputs (additional arguments beside the name) with

`\UseSocket{`⟨*socket-name*⟩`}{`⟨arg_1⟩`}...{`⟨$arg_{\langle number\text{-}of\text{-}inputs \rangle}$⟩`}`

In addition, several code blocks (a.k.a. plugs) implementing different logic for this socket are set up, each with a declaration of the form:

`\NewSocketPlug{`⟨*socket-name*⟩`}{`⟨*socket-plug-name*⟩`}{`⟨*code*⟩`}`

Finally, one of them is assigned to the socket:

`\AssignSocketPlug{`⟨*socket-name*⟩`}{`⟨*socket-plug-name*⟩`}`

If the programming logic should change, then all that is necessary is to make a new assignment with `\AssignSocketPlug` to a different `{`⟨*socket-plug-name*⟩`}`. This assignment obeys scope so that an environment can alter a socket without the need to restore the previous setting manually.

If the socket takes inputs, then those need to be provided to `\UseSocket` and in that case they can be referenced in the ⟨*code*⟩ argument of `\NewSocketPlug` with `#1`, `#2`, etc.

In most cases a named socket is used only in a single place, but there is, of course, nothing wrong with using it in several places, as long as the code in all places is supposed to change in the same way.

2.3.1 Examples

We start by declaring a new socket named `foo` that expects two inputs:

```
\NewSocket{foo}{2}
```

Such a declaration has do be unique across the whole LaTeX run. Thus, if another package attempts to use the same name (regardless of the number of inputs) it will generate an error:

```
\NewSocket{foo}{2}
\NewSocket{foo}{1}
```

Both declarations would therefore produce:

```
! LaTeX socket Error: Socket 'foo' already declared!
```

You also get an error if you attempt to declare some socket plug and the socket name is not yet declared, e.g.,

```
\NewSocketPlug{baz}{undeclared}{some code}
```

generates

```
! LaTeX socket Error: Socket 'baz' undeclared!
```

Setting up plugs for the socket is done like this:

```
\NewSocketPlug{foo}{plug-A}
   {\begin{quote}\itshape foo-A: #1!#2\end{quote}}
\NewSocketPlug{foo}{plug-B}
   {\begin{quote}\sffamily foo-B: #2\textsuperscript{2}\end{quote}}
```

This will set up the plugs `plug-A` and `plug-B` for this socket.

We still have to assign one or the other to the socket, thus without doing that the line

```
\UseSocket{foo}{hello}{world}
```

produces nothing because the default plug for sockets with 2 inputs is `noop` (which grabs the additional arguments and throws them away).[2]

So let's do the assignment

```
\AssignSocketPlug{foo}{plug-A}
```

and then

```
\UseSocket{foo}{hello}{world}
```

will properly typeset

foo-A: hello!world

and after

```
\AssignSocketPlug{foo}{plug-B}
```

and another call to

```
\UseSocket{foo}{hello}{world}
```

we get

foo-B: world[2]

If we attempt to assign a plug that was not defined, e.g.,

```
\AssignSocketPlug{foo}{plug-C}
```

then we get an error during the assignment

```
! LaTeX socket Error: Plug 'plug-C' for socket 'foo' undeclared!
```

and the previous assignment remains in place.

To see what is known about a socket and its plugs you can use `\ShowSocket` or `\LogSocket` which displays information similar to this on the terminal or in the transcript file:

```
Socket foo:
    number of inputs = 2
    available plugs = noop, plug-A, plug-B
    current plug = plug-B
    definition = \long macro:#1#2->\begin {quote}\sffamily
foo-B: #2\textsuperscript {2}\end {quote}
```

[2] If socket `foo` would have been a socket with one input, then the default plug would be `identity`, in which case the socket input would remain without braces and gets typeset!

2.3.2 Details and semantics

In this section we collect some normative statements.

- From a functional point of view sockets are like simple TEX macros, i.e., they expect 0 to 9 mandatory arguments (the socket inputs) and get replaced by their "expansion"
- A socket is "named" and the name consists of ASCII letters `[a-z]`, `[A-Z]`, `[0-9]`, `[-/@]` only
- Socket names have to be unique, i.e., there can be only one socket named ⟨*name*⟩. This is ensured by declaring each socket with `\NewSocket`.

  However, there is no requirement that sockets and hook names have to be different. In fact, if a certain action that could otherwise be specified as hook code has to be executed always last (or first) one could ensure this by placing a socket (single action) after a hook (or vice versa) and using the same name to indicate the relationship, e.g.,

  ```
  \UseHook{foo}        % different package can add code here
  \UseSocket{foo}      % only one package can assign a plug
  ```

  This avoids the need to order the hook code to ensure that something is always last.
- Best practice naming conventions are ... *to be documented*
- A socket has documented inputs which are
  - the positional arguments (if any) with a description of what they contain when used
  - implicit data (registers and other 2e/expl3 data stores) that the socket is allowed to make use of, with a documented description of what they contain (if relevant for the task at hand—no need to describe the whole LATEX universe)
  - information about the state of the TEX engine (again when relevant), e.g. is called in mmode or vmode or in the output routine or ...
  - ... anything missing?
- A socket has documented results/outputs which can be
  - what kind of data it should write to the current list (if that is part of its task)
  - what kind of registers and other 2e/expl3 data stores it should modify and in what way
  - what kind of state changes it should do (if any)
  - *... anything else?*
- At any time a socket has one block of code (a plug :-)) associated with it. Such code is itself named and the association is done by linking the socket name to the code name (putting a plug into the socket).
- The name of a plug consists of ASCII letters `[a-z]`, `[A-Z]`, `[0-9]`, `[-/@]` only.

- Socket plug names have to be unique within on a per socket basis, but it is perfectly allowed (and sensible in some cases) to use the same plug name with different sockets (where based on the sockets' purposes, different actions may be associated with the plug name). For example `noop` is a plug name declared for every socket, yet it action "grab the socket inputs and throw them away" obviously differs depending on how many inputs the socket has.
- When declaring a plug it is stated for which socket it is meant (i.e., its code can only be used with that socket). This means that the same plug name can be used with different sockets referring to different code in each case.
- Configuration of a socket can only be done by linking different code to it. Nevertheless the code linked to it can provide its own means of configuration (but this is outside of the spec).
- Technically execution of a socket (`\UseSocket`) involves
  - doing any house keeping (like writing debugging info, ...);
  - looking up the current code association (what plug is in the socket);
  - executing this code which will pick up the mandatory arguments (happens at this point, not before), i.e., it is like calling a csname defined with

    ```
    \def\foo#1#2...{...#1...#2...}
    ```

  - do some further house keeping (if needed).
- A socket is typically only used in one place in code, but this is not a requirement, i.e., if the same operation with the same inputs need to be carried out in several places the same named socket can be used.

2.3.3 Command syntax

We give both the LaTeX 2_ε and the L3 programming layer command names.

`\NewSocket` `\NewSocket {⟨socket-name⟩} {⟨number-of-inputs⟩}`
`\socket_new:nn` `\socket_new:nn {⟨socket-name⟩} {⟨number-of-inputs⟩}`

Declares a new socket with name ⟨*socket-name*⟩ having ⟨*number-of-inputs*⟩ inputs. There is automatically a plug `noop` declared for it, which does nothing, i.e., it gobbles the socket inputs (if any). This is made the default plug except for sockets with one input which additionally define the plug `identity` and assign that as their default.

This `identity` plug simply returns the socket input without its outer braces. The use case for this plug are situations like this:

`\UseSocket{tagsupport/footnote}{`⟨*code*⟩`}`

If tagging is not active and the socket contains the plug `identity` then this returns ⟨*code*⟩ without the outer braces and to activate tagging all that is necessary is to change the plug to say `tagpdf` so that it surrounds ⟨*code*⟩ by some tagging magic. This is the most common use case for sockets with one input, which is why they have this special default.

The socket documentation should describe its purpose, its inputs and the expected results as discussed above.

The declaration is only allowed at top-level, i.e., not inside a group.

\NewSocketPlug
\socket_new_plug:nnn
\socket_set_plug:nnn

```
\NewSocketPlug       {⟨socket-name⟩} {⟨socket-plug-name⟩} {⟨code⟩}
\socket_new_plug:nnn {⟨socket-name⟩} {⟨socket-plug-name⟩} {⟨code⟩}
\socket_set_plug:nnn {⟨socket-name⟩} {⟨socket-plug-name⟩} {⟨code⟩}
```

Declares a new plug for socket ⟨*socket-name*⟩ that runs ⟨*code*⟩ when executing. It complains if the plug was already declared previously.

The form `\socket_set_plug:nnn` changes an existing plug. As this should normally not be necessary, we currently have only an L3 layer name for the few cases it might be useful.

The declarations can be made inside a group and obey scope, i.e., they vanish if the group ends.

\AssignSocketPlug
\socket_assign_plug:nn

```
\AssignSocketPlug      {⟨socket-name⟩} {⟨socket-plug-name⟩}
\socket_assign_plug:nn {⟨socket-name⟩} {⟨socket-plug-name⟩}
```

Assigns the plug ⟨*socket-plug-name*⟩ to the socket ⟨*socket-name*⟩. It errors if either socket or plug is not defined.

The assignment is local, i.e., it obeys scope.

\UseSocket
\socket_use:nw
\socket_use:n
\socket_use:nn
\socket_use:nnn
\socket_use:nnnn

```
\UseSocket      {⟨socket-name⟩}
\socket_use:nnn {⟨socket-name⟩} {⟨socket-arg1⟩} {⟨socket-arg2⟩}
```

Executes the socket ⟨*socket-name*⟩ by retrieving the ⟨*code*⟩ of the current plug assigned to the socket. This is the only command that would appear inside macro code in packages.

For performance reasons there is no explicit check that the socket was declared!

The different L3 programming layer commands are really doing the same thing: they grab as many arguments as defined as inputs for the socket and then pass them to the plug. The different names are only there to make the code more readable, i.e., to indicate how many arguments are grabbed in total (note that no runtime check is made to verify that this is actually true). We only provide them for sockets with up to 3 inputs (most likely those with zero or one input would have been sufficient). If you happen to have a socket with more inputs, use `\socket_use:nw`.

\socket_use_expandable:nw ⋆
\socket_use_expandable:n ⋆

```
\socket_use_expandable:n {⟨socket-name⟩}
```

Fully expandable variant of `\socket_use:n`. This can be used in macro code to retrieve code from sockets which need to appear in an expandable context.

This usually requires the plug to only contain expandable code and should therefore only be used for sockets which are clearly documented to be used in an expandable context. This command does not print any debugging info when `\DebugSocketsOn` is active and should therefore be avoided whenever possible.

For performance reasons there is no explicit check that the socket was declared!

\ShowSocket
\LogSocket
\socket_show:n
\socket_log:n

```
\ShowSocket    {⟨socket-name⟩}
\socket_show:n {⟨socket-name⟩}
```

Displays information about the socket ⟨*socket-name*⟩ and its state then stops and waits for further instructions — at the moment some what rudimentary.

`\LogSocket` and `\socket_log:n` only differ in that they don't stop.

\DebugSocketsOn
\DebugSocketsOff
\socket_debug_on:
\socket_debug_off:

`\DebugSocketsOn ... \DebugSocketsOff`

Turns debugging of sockets on or off.

2.3.4 Rationale for error handling

The errors during the declarations are produced to help with typos—after all, such declarations might be part of a document preamble (not that likely, but possible). However, `\UseSocket` is not doing much checking, e.g.,

```
\UseSocket{mispelled-socket}{hello}{world}
```

will generate a rather low-level error and then typesets "helloworld" because there is no dedicated runtime check if `mispelled-socket` is a known socket.

The reason is that if the misspelling is in the code, then this is a programming error in the package and for speed reasons LaTeX does not repeately make runtime checks for coding errors unless they can or are likely to be user introduced.

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.

A

\AddToHook 2
\AssignSocketPlug 8
\AtBeginDocument 2

D

\DebugSocketsOff 9
\DebugSocketsOn 8

L

\LogSocket 8

N

\newpage 1
\NewSocket 7
\NewSocketPlug 4

S

\ShowSocket 8
socket commands:
\socket_assign_plug:nn 8
\socket_debug_off: 9
\socket_debug_on: 9
\socket_log:n 8
\socket_new:nn 7
\socket_new_plug:nnn 8
\socket_set_plug:nnn 8
\socket_show:n 8
\socket_use:n 8
\socket_use:nn 8
\socket_use:nnn 8
\socket_use:nnnn 8
\socket_use:nw 8
\socket_use_expandable:n 8
\socket_use_expandable:nw 8

U

\UseSocket 7