PiTEX

Paul Isambert
zappathustra@free.fr
November 2011

1 Introduction

If you're reading this, either you saw `\input pitex` at the beginning of the documentation of one of my packages, or you spend desperate hours browsing CTAN, or you're Arnaud Schmittbuhl. In any case, you're welcome to use the PiTEX macros, provided that you don't forget that: nothing is guaranteed; changes might occur without warning nor retrocompatibility; the documentation isn't necessarily up-to-date; and, if you still want to, you must load PiTEX with `\input pitex` on top of plain TEX, using LuaTEX.

What is PiTEX? Originally, it was a set of files I loaded with plain TEX to typeset documentation for my packages. But it's not just a few macros anymore, but rather a format in progress. The format might never see the public light, but if it does, its originality (compared to existing format) will be an organization based on the Gates package: everything will be highly customizable, not because there are tons of options (although that can be the case too), but because big operations are divided into gates, i.e. macros with a handle that you can control without having to rewrite (nor understand) the big operation you're modifying; which big operation also keeps its integrity, because removing or adding something is neatly done. It also means that PiTEX can be changed piecewise, and made into something that bears little resemblance to the original code. In other words, there is nothing private, nothing forbidden. For the moment, only sectionning commands and the output routine, plus callback management in Lua and `\everypar`, are written with gates. See the Gates documentation for further information.

Another thing with PiTEX is that it will rely heavily on external packages. There will be as little PiTEX-only code as possible. Rather, in line with Gates, each area will be independant code able to work with other formats. That is no simple task, though, and far from complete. For instance, the user's interface is made with YaX, which can be used (and is used) elsewhere.

PiTEX uses three types of files: mandatory external packages, i.e. independant code that PiTEX can't do without, optional external packages, i.e. independant code that can be used, but is not automatically loaded, and mandatory PiTEX-only files. All mandatory files aren't necessary to the same degree, though, and in the future switches might be available to load only some of them. Currently, the files are:

Mandatory external packages

texapi	Macros to write independant code.
YaX	User's interface (and convenient programming tool) with `key = value` style).
Gates	Overall architecture for modular code.

Navigator	PDF features (links, bookmarks…) Currently used by sections, but might become non-mandatory (although strongly recommended).

Mandatory PiTEX files

pitex.tex	The main file that inputs the other, and contains a few macros.
lua.ptx	Lua-related macros.
base.ptxlua	Lua functions, input in the previous file.
files.ptx	Dealing with files.
fonts.ptx	Interface for fonts ; relies on the next file.
fonts.ptxlua	The Lua fontloader ; should become independant some day.
foundry-settings.lua	Default settings for the fontloader.
sections.ptx	Sectionning commands.
blocks.ptx	Blocks (delimited text with special formatting).
references.ptx	Labels and references.
verbatim.ptx	Typesetting verbatim material.
inserts.ptx	Footnotes and figures.
output.ptx	Page layout and output routine.

The following can be used with PiTEX ; actually I only list the packages I've written, but anything working with plain TEX (e.g. TikZ) works with PiTEX.

Optional external packages

Librarian	To create bibliographies without BibTEX.
Lecturer	For screen presentations.
Interpreter	To write text with non-TEX markup (as this documentation) ; Interpreter does the conversion on the fly.

The PiTEX distribution also contains `i-pitex.lua`, an interpretation file for Interpreter used to typeset documentations, like the one you are currently reading. Which is why you can read it quite comfortably as a plain text file in a text editor (see `pitex-doc.txt`).

The rest of this document is a terse description of existing commands, parameters, and of course, gates.

2 Fonts (fonts.ptx and fonts.ptxlua)

The fontloader uses gates, but only superficially. They won't be documented here.

`\setfont <command>:<attributes>`

Sets `<command>` to call the font described in `<attributes>` ; all defaults to the values of the `metafont` parameter. If `<command>` is `\mainfont`, the font is called at once. Also, `\codefont` is used in some places (e.g. verbatim).

`name`
: The family name of the font; e.g. Chaparral Pro for the main text of this document.

`size (dimension)`
: The size of the font.

`small (dimension)`
: The size of the font when `\small` is called. Can be a relative value by prefixing it with - or +, in which case it is set relative to `size`.

`verysmall (dimension)`
: The size (possibly relative) for `\verysmall`.

`big (dimension)`
: The size (possibly relative) for `\big`.

`verybig (dimension)`
: The size (possibly relative) for `\verybig`.

`bold (font modifier)`
: The modifier used for the bold version of the font, without the leading slash; `metafont` sets it to `Bold`.

`italic (font modifier)`
: Same as `bold` for the italic version; set to `Italic` by `metafont`.

`math (true or false)`
: If true, math fonts will be created.

`features`
: Well, err, font features...

`slant (angle)`
: The slant applied to the font to create a fake italic.

`slantsc (angle)`
: The slant applied to the font to create fake italic smallcaps; if not given, defaults to `slant`.

  There's actually much more going under the hood, but `font.ptxlua` (the font loader itself) is a work in progress, and undocumented.

  The same macros as in plain TEX can be used, except they're cumulative, i.e. `\it\bf` switches to a bold italic.

`\it`
: Switches to italics.

`\rm`
: Switches to roman.

`\bf`
: Switches to bold.

`\rg`
: Switches to regular weight.

`\sc`
: Switches to small capitals.

`\lc`
: Switches to lower case (i.e. not small caps).

`\ital <text>`
: Typesets `<text>` in italics.

`\bold <text>`
: Typesets `<text>` in bold.

`\scap <text>`
: Typesets `<text>` in small caps.

`\rom <text>`
: Typesets `<text>` in roman.

`\emph <text>`
: Typesets `<text>` in italics or roman, depending on whether the current font is roman or italics, respectively.

`\underline <text>`
: Underlines `<text>`. Wow.

`\small`
: Switches to small font.

`\verysmall`
: Switches to very small font.

`\big`
: Switches to big font.

`\verybig`
: Got it?

`\normalsize`
: Switches to default size.

`\smaller`
: Switches to the font smaller than the current one (e.g. `\normalsize` if you're currently using `\big`).

`\bigger`
: Same as `\smaller`, the other way around.

`\color <color> <text>`
: Typesets `<text>` with `<color>`, which should be a triplet `R G B` with each value between 0 and 1.

3 Sections (sections.ptx)

Sections are among the victims of my fanaticism for grid-typesetting.

3.1 Main sectioning commands

`\declaresection <type> <level>`
: Creates a new section type with `<level>`. This is not necessary to make `\sectioncommand` work with `<type>`, but with it all declared sections of level larger than `<level>` are reset (i.e. their counters are set to 0). Sections with `<type>` chapter, section, subsection and paragraph are already declared.

`\incrementsection <type>`
: Increments the counter of section <type>. If <type> hasn't been declared, a new <type> is created, but without a level.

`\getsectioncounter <type>`
: Returns the value of section <type>'s counter, or -1 if there is no section of that type.

`\sectioncommand <type><title><alternate title>[<label>]`
: Creates a section heading of type <type> with <title>. See the details of the gates involved below. The <type> refers to the parameter of the same name. The <alternate title> is very likely to disappear.

`\chapter <title>[<label>]`
: Equivalent to `\sectioncommandchapter{<title>}{<title>}[<label>]`.

`\section <title>[<label>]`
: Equivalent to `\sectioncommandsection{<title>}{<title>}[<label>]`.

`\subsection <title>[<label>]`
: Equivalent to `\sectioncommandsubsection{<title>}{<title>}[<label>]`.

`\paragraph <title>[<label>]`
: Equivalent to `\sectioncommandparagraph{<title>}{<title>}[<label>]`.

`\ifsectiontitle`
: A conditional that is true when the section title is being typeset (sets by the section_pre gate below.

`\sectioninfile <optional star><title><space><type><space><file><space>`
: Creates a section with the contents of a file, unless there's an optional star (useful to typeset only parts of a big document); <title> is a `\freedef` argument, hence it can be given between braces, double quotes or slashes (but the <space> is nonetheless mandatory); <type> is a section type, and <file> is a file to be input: it shouldn't have an extension (tex files are searched), but it can be a path with / as a separator. A `\label` is also created, with the tail of <file> as its argument. In other words, the following:

```
\sectioninfile "A chapter" chapter mydir/myfile
```

is equivalent to:

```
\chapter{A chapter}
\label{myfile}
\input mydir/myfile.tex
```

The `\sectioncommand` macro only contains a list gate, `section`, itself containing the gates typesetting a section heading; all the gates belong to the `Section` family associated with the `\Section` command. Here are all the gates involved; the first number between parentheses indicates how many arguments the gate should receive, the second how many it returns.

section	(4, 0)	section_break	(1, 0)	section_vmode	(1, 0)
				section_clearpage	(1, 0)
				section_beforeskip	(1, 0)
		section_advance	(1, 0)		
		section_advance	(1, 0)		
		section_bookmark	(4, 0)		
		section_toc	(3, 0)		
		section_pre	(0, 0)		
		section_typeset	(2, 2)	section_number	(2, 3)
				section_heading	(3, 2)
				section_addfont	(2, 2)
				section_addcolor	(2, 2)
				section_do	(2, 0)
		section_post	(0, 0)		
		section_afterskip	(1, 0)		

Here's how the gates work:

`section <type><title><alternate title><label>`
: The main list gate that contains all sections. In what follows, when I mention an attribute, I mean the attribute of the parameter <type>.

`section_break <type>`
: An l-gate managing whatever must happen before the section title is typeset.

`section_vmode <type>`
: Inserts a `\par` and removes last skip. (Conditional: `vmode` is `true`.)

`section_clear <type>`
: Inserts a `\clearpage`. (Conditional: `clear` is `true`.)

`section_beforeskip <type>`
: Creates a vertical skip before the heading. If the current page cannot accommodate `beforeskip` + `minimum` + `afterskip` worth of lines, then the section heading is typeset on the next page (using `\breakpage`). If the current page can accommodate the section, a skip of `beforeskip` lines is inserted. The gate doesn't return anything. (Conditional: `clear` is not `true`.)

`section_advance <type>`
: Increments the section counter, and resets the counters of those sections whose level is larger than <type>'s level (provided <type> has been declared with `\declaresection` and thus given a level). The gate doesn't return anything.

`section_bookmark <type><title><alternate title><label>`
: The bookmark is created with Navigator's `\outline` command as follows:

```
\outline[meta = <type>bookmark]{<bookmarklevel>}[<label>]{<alternate title>}
```

only if `bookmarklevel` is defined. For types chapter, section, subsection and paragraph, the related `chapterbookmark`, `sectionbookmark`, `subsectionbookmark` and `paragraphbookmark`

parameters are predefined, with simply meta set to navigator. The <alternate title> is likely to disappear, since Navigator can handle things correctly. The gate doesn't return anything. (Conditional : link is true.)

section_toc <type><title><alternate title>
Writes what should be written to the auxiliary file for the next run to produce a table of contents. The gate doesn't return anything. (Conditional : toc is true.)

section_pre
Prepares the typesetting : open a group, sets \maintextfalse and \sectiontitletrue, and sets a LuaTEX attribute to 0 (so the section title is marked and can be spotted in the output routine). The gates doesn't return anything.

section_typeset <type><title>
A list gate containing the gates used to typeset the section heading. It returns its final two arguments if only because list gate automatically return. See description below.

section_post
Closes the group opened in section_pre. The gate doesn't return anything.

section_afterskip <type>
Creates a vertical skip of afterskip lines. Also, calls \removenextindent if removenextindent is true. The gate doesn't return anything. (Conditional : inline is not true.)

Here are the gates contained in section_pre. Beware, there the nature of the passed arguments slightly changes.

section_indent <type>
Goes into horizontal mode and inserts an indent of width indent. The gate doesn't return anything.

section_number <type><title>
Sets the section number, if number isn't none. The number is surrounded by beforenumber and afternumber, converted to roman or arabic number according to the value of number, and the whole is passed to numbercommand (if it exists). The gate returns the following three arguments : <type><number><title>, where <number> is what's just been described.

section_heading <type><number><title>
Sets the <title> : it is prefixed with <number>, passed to function if it exists, and suffixed with aftertitle if any. The gate returns <type> and <title> as just described.

section_addfont <type><title>
Prefixes <title> with the value of font and returns its two arguments.

section_addcolor <type><title>
Adds color to the title and returns its two arguments. (Conditional : color is true.)

section_do <type><title>
At last ! Inserts an horizontal skip of width indent, typesets <title>, and if inline isn't true, sets \rightskip to ragged. Oh, yes, this could be divided into smaller gates. The gates doesn't return anything.

The relevant parameters are the one corresponding to the type of the section, i.e. chapter, section, subsection, paragraph, which all have metasection as their meta-parameter. The relevant attributes are :

`vmode` (`true` or `false`)
: If `true`, goes into vertical mode before typesetting the heading.

`clear` (`true` or `false`)
: If `true`, the section starts on a new page.

`beforeskip` (glue)
: The skip added before the heading.

`afterskip` (glue)
: The skip added after the heading.

`minimum` (number)
: The minimum number of lines that should be present on the page after the section heading. The `section_skip` gate above starts a new page if `beforeskip` + `afterskip` + `minimum` can't be accommodated.

`inline` (`true` or `false`)
: If `true`, the section heading is inserted at the beginning of the following paragraph.

`number` (`arabic`, `roman` or `none`)
: The way the section number should be typeset; `none` means the number isn't typeset.

`beforenumber`
: Material to be added before the section number.

`afternumber`
: Material to be added after the section number.

`numbercommand` (control sequence)
: A macro to which the section number (surrounded by `beforenumber` and `afternumber`) is passed.

`function` (control sequence)
: A macro to which the section title is passed.

`aftertitle`
: Material added after the section title.

`font`
: Font for the heading.

`color` (a triplet of values)
: Color for the heading.

`indent` (glue)
: The value of the glue added before the section title.

`ragged` (glue)
: The value of `\rightskip` for the heading.

`toc` (`true` or `false`)
: Sets whether the section should be added to the table of contents or not.

`removenextindent` (`true` or `false`)
: Sets whether the next paragraph should be unindented.

`link` (`true` or `false`)
: Sets whether a bookmark should be created with the section's title.

`bookmarklevel (number)`
: The level of the bookmark created for the section (how surprising). Further specification of the bookmark is done with `chapterbookmark`, `sectionbookmark`, `subsectionbookmark`, `paragraphbookmark`, whose only specification is that `meta` is set to `navigator`. New `<type>bookmark`'s can be created, of course. See the documentation of Navigator for advanced use.

3.2 Various commands

`\tableofcontents`
: Writes the table of contents (needs two runs). Not customizable for the time being!

`\newbreakpenalty <command>`
: Defines `<command>` as a number below `-10000`, suffixed with a `\relax`. The idea is to use it to break a page and check it in the output routine.

`\clearpage`
: Fills the rest of the page with white space.

`\clearpagepenalty`
: Penalty associated with `\clearpage`.

`\breakpage`
: Same as `\clearpage`. They shouldn't be used for the same reasons. I use `\clearpage` at the end of a chapter, and `\breakpage` elsewhere (e.g. when a section heading would be orphaned and must move to the next page). The latter triggers nothing special, but the former can be identified in the output routine and for instance suppress footers.

`\breakpagepenalty`
: Penalty associated with `\breakpage`.

`\needspace <dimen>`
: Moves to the next page if there's less than `<dimen>`.

`\iflines <number><true><false>`
: Executes `<true>` if there's at least `<number>` lines left on the page, and `<false>` otherwise.

`\ignorepars <material>`
: Ignores incoming `\par` commands (and spaces too) and executes `<material>`. Useful when a blank line looks good in the source but you don't want it to signal a paragraph's end. The command is used by sectionning commands, so that if the section's title is supposed to be inserted at the beginning of the next paragraph (e.g. if `inline` is `true`), you can nonetheless leave a blank line after the command.

4 References (references.ptx)

There is a nice reference system, but it is a mess and should be rewritten in Lua. So it isn't fully described here.

`\label <name>`
: Sets a label with `<name>`.

`\ref [<pre>][<post>] <reference type> {<name>}`
: Makes a reference. Beware of the syntax: label should be enclosed between braces, because the

left brace is the delimiter for <reference name>, which in turn should be enclosed in braces. E.g. a call is :

```
\ref page {mylabel}
```

What is returned depends on <reference type>. If it is empty, then what is returned is the value of \ptx@label when \label{<name>} was issued. I think some commands define \ptx@label (nice in blocks, for instance). Otherwise, <reference type> can be page, chapter, section, subsection, paragraph or footnote (the latter if and only if \label was issued in a footnote). The returned text is prefixed with <pre> and suffixed with <post>.

Also, if <reference type> is page, it may take three runs to make things work, because it is checked whether the returned value is the current page, in which case nothing is printed (it's stupid to refer to the current page). As mentioned above, this is a mess.

There also are commands like \sref{label} and the like, which are shorthands for e.g. \ref[section~][] section {label}.

5 Blocks (blocks.ptx)

Blocks are what are called environments elsewhere : they mark up a section of the document, and generally apply some special operations. Given a block myblock, it is launched with \myblock, closed with \myblock/ and continued with \myblock|. As you might imagine, this implies poking at the next token, which in some rare case might be troublesome ; hence, \myblock can be followed by an optional > whose only goal is to protect the next token. (You can also use a \relax, of course.)

\newblock <optional star><command><pre><optional start><optional middle><post>

Defines <command> as a block. If the first optional star is present, the block is executed inside a group. If the second optional star is present, then the <middle> argument should be present too. The block is defined as follow : <pre> is executed at the beginning (i.e. \myblock or \myblock>), <middle> is executed when the block is continued (i.e. \myblock|), and <post> is executed when the block is closed (i.e. \myblock/). If <middle> is not given, then \myblock| does nothing. For instance, the following defines a grouped block (so the \rightskip setting doesn't affect the rest of the document) ; note the \par at the end, so the paragraph is built before the group is closed :

```
\newblock*\raggedblock{\rightskip=0pt plus 1fil\relax}{\par}
```

And here's a simple example with a middle part :

```
\newblock\listblock{\vskip\baselineskip- }
                   *{\par- }
                    {\vskip\baselineskip}
```

The continuation part can be used as a partial block opening: some markers are repeated (the dash) others are not (the vertical space).

`\newblocktype <command><pre><middle><post>`

Defines <command> as a block definition command like \newblock with <pre>, <middle> and <post> to be executed by default before the user-supplied versions. The \newblock command itself has been thus defined, with empty arguments. Arguments, after:

```
\newblocktype\newlist{\vskip\baselineskip-}
                     {\par-}
                     {\vskip\baselineskip}
```

Then \newlist\listblock will have the same definition as in the previous example (no need to supply a <middle> part, it's in the default), and a variation can be created.

`\removenextindent`

Removes the indentation box of the next paragraph (used by section macros). Technically, it sets ajar the noindent gate in the everypar gate list (itself registered in the \everypar token list, which shouldn't be handled otherwise if flexibility is to be ensured). Those two gates belong to the Everypar family associated with the \Everypar command.

`\Indent`

Indents the next paragraph even if it \removenextindent has been issued (a \kern is added).

6 Dealing with files (files.ptx)

`\iffile [<format>]<file><true><false>`

Executes <true> if kpse.find_file (from the LuaTEX kpse library, implementing kpathsea) with file type <format> (default: tex), and <false> otherwise.

`\ifffile [<format>]<file><true>`

Same as \iffile, except nothing happens when the file isn't found. Yes, three *f*'s.

`\inputfileor [<format>]<file><no file>`

Reads file <file> or executes <no file>.

`\writeout <optional star><general text>`

Writes <general text> to the auxilary file that is read at the beginning of each job. Without a star, writing happens at once (it's \immediate), with it writing is delayed until the current page is shipped out.

7 Verbatim (verbatim.ptx)

`\verbcatcodes`

A catcode table with usual verbatim catcodes: special characters have catcode 12, except space and end-of-line, which have catcode 13 and are defined to \quitvmode\spacecs and \quitvmode\par by default.

`\newverbatim <command> [<catcode table>] <pre> <post>`

Defines a new block `<command>` with `<catcode table>`, `<pre>` at the beginning and `<post>` at the end. If `<catcode table>` is missing, `\verbcatcodes` is used. Verbatim blocks work as follows: first, there is no continuation command, i.e. only `\myverbatimblock` and `\myverbatimblock/` are allowed, not `\myverbatimblock|` (it might exist somewhere in the future). Second, the block opening takes one optional argument between brackets, which is the name of the verbatim block. Third, `<pre>` is executed at the beginning, and `<post>` at the end, as defined with `\newverbatim`. Fourth, the end statement `\myverbatimblock/` should be on a line of its own. What a verbatim block does is the following (not taking into account what `<pre>` and `<post>` execute): it stores its contents as is, along with the `<catcode table>` the block was declared with, and that's it. Then come the following two functions.

`\doverbatim [<name>]`

Executes the contents of `<name>` (with the current catcode regime). If name is missing, last is used, a special name which refers to the last verbatim block.

`\printverbatim [<name>]`

Executes the contents of `<name>` (or last if `<name>` is missing) with the catcode table associated with the block `<name>` was stored with. Since that catcode table is `\verbcatcodes` by default, it generally results in the contents being typeset.

As an example:

```
\newverbatim\myverbatim{\vskip\baselineskip}
{\printverbatim\vskip\baselineskip}
\myverbatim[example]
\def\foo{hello !}%
\foo
\myverbatim/
And now we are going to print: \doverbatim[example].
```

`\verbatim`

A predefined verbatim block, designed as follows:

```
\newverbatim\verbatim{\codefont\parindent0pt}
                     {\vskip\baselineskip\printverbatim\relax
                      \vskip\baselineskip\removenextindent}
```

I.e. it switches to the console-like font, sets the paragraph indentation to nothing, prints its contents between two blank lines and removes the indentation of the paragraph to follow.

Each verbatim block adds a table to the Lua table `pitex.verbatims` (yes, with an *s*); the key is the block's name, and the value is a table with lines as values, indexed by numbers, plus a `regime` key which returns the catcode table's number of the block. For instance, the core operation performed by `\printverbatim[<name>]` is:

```
tex.print(pitex.verbatims[<name>].regime, pitex.verbatims[<name>])
```

8 Insertions (inserts.ptx)

Insertions are still a mess, and not related to parameters. Yet you can use:

`\footnote <text>`
: Typesets `<text>` in a footnote. How astounding.

`\figure [<title>]`
: A block creating a figure with title `<title>`.

`\table [<title>]`
: The same as `<figure>`, except `Table` will be used instead of `Fig` in the caption.

`\infigure`
: Ablock creating a figure in the main text, i.e. between paragraphs.

9 Layout and output routine (output.ptx)

The page layout can be specified with the `page` parameter, whose attributes are:

`width (dimension)`
: The width of the page.

`height (dimension)`
: The height of the page.

`baselineskip (glue)`
: The baseline distance.

`topskip (glue)`
: The distance between the top of the textblock and the first baseline.

`top (dimension)`
: The height of the upper margin.

`lines`
: The number of lines on a page.

`hsize (dimension)`
: The width of the textblock.

`left (dimension)`
: Width of the left margin.

`right (dimension)`
: Width of the right margin. If specified, `hsize` is ignored and the texblock's width is set to `width - left - right`.

`parindent (dimension)`
: The width of the indentation.

`parskip (glue)`
: The glue between paragraphs.

The output routine holds nothing very interesting for the moment. I used to redefine it for each job. Now it is set up with gates, but I haven't taken the time yet to make it really powerful.

Plus I should rewrite everything in Lua as much as possible. Anyway, `\output` contains the `output` gate, from the `OutputRoutine` family associated with the `\OutputRoutine` command; the gates work as follows (passed arguments aren't indicated, because there isn't any; although someday perhaps the gates will pass a box between them, to be less dependant on `\outputbox`, which is used, by the way, instead of box 255, so any box register can be used):

output
- precheck
- shipout
  - processmarginalia
  - inserts
    - inserts_figures
    - inserts_footnotes
  - headers
  - ship
  - postship

output
The main list gate, holding the following.

precheck
Checks whether `\outputpenalty` is smaller than `\widowpenalty`. If not, `\vsize` is increased or decreased (if there are inserts) by `\baselineskip`, so that the widow is accommodated or a line is given to the next page. In any case, the output box is repassed to TEX with `\holdinginserts=0`. The gate is then set to `skip`, so it isn't executed again.

shipout
A list gate containing gates to write the page. By default it is skipped, so it isn't executed when the previous gate is, and vice-versa.

processmarginalia
Insert the marginal notes (see `\marginnote` below). Can be obviously removed if there are no such notes.

inserts
An l-gate containing the following two m-gates.

inserts_figures
Adds the figures. (Conditional: the box `\ptx@insert_figures` isn't empty.)

inserts_footnotes
Adds the footnotes. (Conditional: the box `\ptx@insert_footnotes` isn't empty.)

headers
Inserts headers or footers, i.e. page number, running title, etc.

ship
Ships out the page.

postship
Resets some stuff (set `output_shipout` back to `skip`), and increments the page number.

And now a lonely command:

`\marginnote [options] <text>`
Produces a marginal note with `<text>`. Uses the attributes (font, baselineskip, hsize) of the `marginnote` parameter, with the following attributes:

hsize (dimension)
: Width of the textblock in the note.

hpos (ff, fr, rf, rr)
: Justification of the text: flushed on both sides, ragged on the right, ragged on the left, ragged on both sides.

font
: Font used to typeset the note.

parindent (dimension)
: Paragraph indentation for the note.

side (left or right)
: Side of the note relative to the textblock. That should depend on whether the note is on an odd or even side, but for the moment that is not the case.

gap (dimension)
: Distance between the textblock and the note.

10 Lua facilities (lua.ptx and base.ptxlua)

\inputluafile <file>
: Shorthand for \directluadofile(kpse.find_file(<file>)).

\luacatcodes
: A catcode table with Lua-convenient catcodes: #, ~, % and the end of file ^^M are set to catcode 12.

\luacode
: A block to write Lua code with the catcodes above.

Lua code in PiTEX is organized mostly in gates; pitex is a gate table associated with family pitex, pitex.callback is another table associated with family pitex.callback, and pitex.misc is a third table associated (how surprising) with family pitex.misc. The division of labour isn't perfectly defined, to say the least.

The pitex family holds general commands, namely:

pitex.log (<message>, ...)
: Writes a <message> formatted as string.format(<message>, ...)

pitex.error (<message>, ...)
: Same as the previous function, but less friendly.

The pitex.callback family is concerned with callback management. It has one interesting function (well, a gate) devoted to handling functions in callbacks as gates:

pitex.callback.register (<callback>, <gate>)
: An l-gate is registered in the callback, with subgates added to it; the name of the l-gate is the same as the callback where it is registered (with the family prefix pitex.callback added when necessary). For instance the list gate containing functions to used in process_input_buffer is called process_input_buffer. Ordinary you would add a subgate to such a callback with the add action:

```
pitex.callback.add ("mygate", "process_input_buffer")
```

However, the l-gate associated with the callback isn't created by default, nor registered in the callback. This means that `add` above will fail miserably if `process_input_buffer` hasn't been created beforehand. This function is meant to circumvent that: if the l-gate exists, it boils down to `add`; otherwise it creates it and registers it in the callback, and then add the gate. Note that the syntax follows the original `callback.register` function, with the callback first and the function second, even though you're adding gates to l-gates, with the syntax of `add` being subgates first, l-gate second. To manage the gates, thus created, you can then rely on the original gate actions.

So, when I say 'gates `X` and `Y` are registered in callback `Z`', it means 'gates `X` and `Z` are subgates of l-gate `pitex.callback:Z`, itself registered in callback `Z`'; unless otherwise indicated, `X` and `Z` belong to the `pitex.callback` family.

And here are the gates registered in callbacks: `process_input_buffer` contains `convert`, which turns `latin1` into UTF-8. Verbatim blocks also register `process_verbatim`, which is removed when the block ends. The `kerning` callback contains `french_punctuation`, meant to add thin space before some punctuation mark, and `original_kerning`, which is just a gate version of the `node.kerning` function. In `post_linebreak_filter` you'll find `pitex.misc:underline`, which deals with material to be underline and `pitex.misc:mark_lines`, which marks lines where a margin note is to be added. Those last two gates should be rewritten as complex l-gates (they're just big functions for the moment) some day. If you neither underline nor use marginal notes, you can remove them.

11 Things that didn't make it elsewhere (pitex.tex)

General properties of the document can be set with the `document` parameter, with the following attributes (the `navigator` parameter has a `meta` attribute set to `document`, which is why you'll find attributes here used by Navigator):

`author`
: The author of the document.

`title`
: The title of the document.

`pdftitle`
: The title that Navigator will use for the document's properties (defaults to `title`).

`date`
: The date of the document

`pdfdate`
: The date that Navigator will use for the document's properties; should be a properly formatted PDF date (this corresponds to the `date` attribute in the `navigator` parameter; note that `pdfdate` doesn't defaults to `date`, because the latter is supposed to hold a readable date).

`subject`
: For the document's properties.

`keywords`
: Again, the properties.

mode (outlines, bookmarks, thumbs, thumbnails, attachments, files, oc)
: What should be displayed in the navigation bar when the document is opened. See the documentation to Navigator.

layout (onepage, onecolumn, twopage, twocolumn, twopage*, twocolumn*)
: How the document is displayed when opened. See the documentation of Navigator.

\newattribute <command>
: Defines <command> as an attribute register.

\unsetattribute <command>
: Unsets attribute <command>.

\attributenumber <command>
: Returns the number of attribute register <command> (not its value ; you get the value with \the ; this is to pass to Lua).

\freedef <command>{<definition>}
: Same as \def\foo#1{...}, except <command> can take its argument between double quotes (") or slahes (/), and of course as single token or brace-delimited.

\ifmaintext
: Conditional that is true when in main text ; inserts, section headings, etc., should turn it to false, and sets their own to true.

\newcatcodetable <command><catcode settings>
: Defines <command> as a catcode table with <catcode settings> ; the latter are <list of characters> = catcode pairs, separated by commas, like the argument of *texapi*'s \setcatcodes.

\texcatcodes
: A code catcode table with the traditional catcodes.

\inputpitexfile <file><space>
: Inputs <file> unless the initialization script says otherwise. If <file> has no extension, .ptx is used.

\antigobblespace
: Adds a space if the next character has catcode 11 or is an opening parenthesis. For instance, after \def\tex{\TeX\antigobblespace}, you can type \tex is typesetting program without worrying for gobbled space. Note that only ASCII letters have catcode 11 by default (not accented characters).

\strut <height><depth>
: Produces an invisible vertical rule with the specified dimensions.

\colorbox [<dimensions>]<RGB color><text>
: Puts <text> in a colored box with background color <RGB color> (e.g. three space-separated numbers between 0 and 1) and with padding <dimensions>. If <dimensions> is missing, padding is done according to the \extraboxspace length. Otherwise, if <dimensions> contains one value, it is used on all side ; if there are two values (separated by commas), the first is used for top and bottom padding, and the second for left and right padding ; with three value, the third specifies bottom padding, and a fourth specifies left padding. Very unlikely to remain in its present state or to remain at all.

`\extraboxspace`
: Default padding for the previous command.

`\og`
: Produces an opening guillemet : « (character `0x00AB`).

`\fg`
: Produces a closing guillemet : » (character `0x00BB`). Uses `\antigobblespace`.

`\trace`
: Sets `\tracingcommands` to 3 and `\tracingmacros` to 2.

`\untrace`
: Sets `\tracingcommands` and `\tracingmacros` to 0.