# PerlTeX: Defining LaTeX macros in terms of Perl code[*]

Scott Pakin
scott+pt@pakin.org

June 24, 2006

**Abstract**

PerlTeX is a combination Perl script (`perltex.pl`) and LaTeX 2$_\varepsilon$ style file (`perltex.sty`) that, together, give the user the ability to define LaTeX macros in terms of Perl code. Once defined, a Perl macro becomes indistinguishable from any other LaTeX macro. PerlTeX thereby combines LaTeX's typesetting power with Perl's programmability.

## 1 Introduction

TeX is a professional-quality typesetting system. However, its programming language is rather hard to use for anything but the most simple forms of text substitution. Even LaTeX, the most popular macro package for TeX, does little to simplify TeX programming.

Perl is a general-purpose programming language whose forte is in text manipulation. However, it has no support whatsoever for typesetting.

PerlTeX's goal is to bridge these two worlds. It enables the construction of documents that are primarily LaTeX-based but contain a modicum of Perl. PerlTeX seamlessly integrates Perl code into a LaTeX document, enabling the user to define macros whose bodies consist of Perl code instead of TeX and LaTeX code.

As an example, suppose you need to define a macro that reverses a set of words. Although it sounds like it should be simple, few LaTeX authors are sufficiently versed in the TeX language to be able to express such a macro. However, a word-reversal function is easy to express in Perl: one need only `split` a string into a list of words, `reverse` the list, and `join` it back together. The following is how a `\reversewords` macro could be defined using PerlTeX:

```
\perlnewcommand{\reversewords}[1]{join " ", reverse split " ", $_[0]}
```

[*]This document corresponds to PerlTeX v1.3, dated 2006/06/24.

Then, executing "`\reversewords{Try doing this without Perl!}`" in a document would produce the text "Perl! without this doing Try". Simple, isn't it?

As another example, think about how you'd write a macro in LaTeX to extract a substring of a given string when provided with a starting position and a length. Perl has an built-in `substr` function and PerlTeX makes it easy to export this to LaTeX:

```
\perlnewcommand{\substr}[3]{substr $_[0], $_[1], $_[2]}
```

`\substr` can then be used just like any other LaTeX macro—and as simply as Perl's `substr` function:

```
\newcommand{\str}{superlative}
A sample substring of ``\str'' is ``\substr{\str}{2}{4}''.
```

$$\Downarrow$$

A sample substring of "superlative" is "perl".

To present a somewhat more complex example, observe how much easier it is to generate a repetitive matrix using Perl code than ordinary LaTeX commands:

```
\perlnewcommand{\hilbertmatrix}[1]{
  my $result = '
\[
\renewcommand{\arraystretch}{1.3}
';
  $result .= '\begin{array}{' . 'c' x $_[0] . "}\n";
  foreach $j (0 .. $_[0]-1) {
    my @row;
    foreach $i (0 .. $_[0]-1) {
      push @row, ($i+$j) ? (sprintf '\frac{1}{%d}', $i+$j+1) : '1';
    }
    $result .= join (' & ', @row) . " \\\\\n";
  }
  $result .= '\end{array}
\]';
  return $result;
}

\hilbertmatrix{20}
```

$$\Downarrow$$

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $1$ | $\frac{1}{2}$ | $\frac{1}{3}$ | $\frac{1}{4}$ | $\frac{1}{5}$ | $\frac{1}{6}$ | $\frac{1}{7}$ | $\frac{1}{8}$ | $\frac{1}{9}$ | $\frac{1}{10}$ | $\frac{1}{11}$ | $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ |
| $\frac{1}{2}$ | $\frac{1}{3}$ | $\frac{1}{4}$ | $\frac{1}{5}$ | $\frac{1}{6}$ | $\frac{1}{7}$ | $\frac{1}{8}$ | $\frac{1}{9}$ | $\frac{1}{10}$ | $\frac{1}{11}$ | $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ |
| $\frac{1}{3}$ | $\frac{1}{4}$ | $\frac{1}{5}$ | $\frac{1}{6}$ | $\frac{1}{7}$ | $\frac{1}{8}$ | $\frac{1}{9}$ | $\frac{1}{10}$ | $\frac{1}{11}$ | $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ |
| $\frac{1}{6}$ | $\frac{1}{7}$ | $\frac{1}{8}$ | $\frac{1}{9}$ | $\frac{1}{10}$ | $\frac{1}{11}$ | $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ |
| $\frac{1}{7}$ | $\frac{1}{8}$ | $\frac{1}{9}$ | $\frac{1}{10}$ | $\frac{1}{11}$ | $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ | $\frac{1}{21}$ |
| $\frac{1}{8}$ | $\frac{1}{9}$ | $\frac{1}{10}$ | $\frac{1}{11}$ | $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ | $\frac{1}{21}$ | $\frac{1}{22}$ |
| $\frac{1}{9}$ | $\frac{1}{10}$ | $\frac{1}{11}$ | $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ | $\frac{1}{21}$ | $\frac{1}{22}$ | $\frac{1}{23}$ |
| $\frac{1}{10}$ | $\frac{1}{11}$ | $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ | $\frac{1}{21}$ | $\frac{1}{22}$ | $\frac{1}{23}$ | $\frac{1}{24}$ |
| $\frac{1}{11}$ | $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ | $\frac{1}{21}$ | $\frac{1}{22}$ | $\frac{1}{23}$ | $\frac{1}{24}$ | $\frac{1}{25}$ |
| $\frac{1}{12}$ | $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ | $\frac{1}{21}$ | $\frac{1}{22}$ | $\frac{1}{23}$ | $\frac{1}{24}$ | $\frac{1}{25}$ | $\frac{1}{26}$ |
| $\frac{1}{13}$ | $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ | $\frac{1}{21}$ | $\frac{1}{22}$ | $\frac{1}{23}$ | $\frac{1}{24}$ | $\frac{1}{25}$ | $\frac{1}{26}$ | $\frac{1}{27}$ |
| $\frac{1}{14}$ | $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ | $\frac{1}{21}$ | $\frac{1}{22}$ | $\frac{1}{23}$ | $\frac{1}{24}$ | $\frac{1}{25}$ | $\frac{1}{26}$ | $\frac{1}{27}$ | $\frac{1}{28}$ |
| $\frac{1}{15}$ | $\frac{1}{16}$ | $\frac{1}{17}$ | $\frac{1}{18}$ | $\frac{1}{19}$ | $\frac{1}{20}$ | $\frac{1}{21}$ | $\frac{1}{22}$ | $\frac{1}{23}$ | $\frac{1}{24}$ | $\frac{1}{25}$ | $\frac{1}{26}$ | $\frac{1}{27}$ | $\frac{1}{28}$ | $\frac{1}{29}$ |

In addition to `\perlnewcommand` and `\perlrenewcommand`, PerlTEX supports `\perlnewenvironment` and `\perlrenewenvironment` macros. These enable environments to be defined using Perl code. The following example, a `spreadsheet` environment, generates a `tabular` environment plus a predefined header row. This example would have been much more difficult to implement without PerlTEX:

```
\newcounter{ssrow}
\perlnewenvironment{spreadsheet}[1]{
  my $cols = $_[0];
  my $header = "A";
  my $tabular = "\\setcounter{ssrow}{1}\n";
  $tabular .= '\newcommand*{\rownum}{\thessrow\addtocounter{ssrow}{1}}' . "\n";
  $tabular .= '\begin{tabular}{@{}r|*{' . $cols . '}{r}@{}}' . "\n";
  $tabular .= '\\multicolumn{1}{@{}c}{} &' . "\n";
  foreach (1 .. $cols) {
    $tabular .= "\\multicolumn{1}{c";
    $tabular .= '@{}' if $_ == $cols;
    $tabular .= "}{" . $header++ . "}";
    if ($_ == $cols) {
      $tabular .= " \\\\ \\cline{2-" . ($cols+1) . "}"
    }
    else {
      $tabular .= " &";
    }
    $tabular .= "\n";
  }
  return $tabular;
}{
  return "\\end{tabular}\n";
}

\begin{center}
  \begin{spreadsheet}{4}
    \rownum &  1 &  8 & 10 & 15 \\
    \rownum & 12 & 13 &  3 &  6 \\
    \rownum &  7 &  2 & 16 &  9 \\
    \rownum & 14 & 11 &  5 &  4
  \end{spreadsheet}
\end{center}
```

$$\Downarrow$$

|   | A | B | C | D |
|---|---|---|---|---|
| 1 | 1 | 8 | 10 | 15 |
| 2 | 12 | 13 | 3 | 6 |
| 3 | 7 | 2 | 16 | 9 |
| 4 | 14 | 11 | 5 | 4 |

## 2 Usage

There are two components to using PerlTeX. First, documents must include a "`\usepackage{perltex}`" line in their preamble in order to define `\perlnewcommand`, `\perlrenewcommand`, `\perlnewenvironment`, and `\perlrenewenvironment`. Second, LaTeX documents must be compiled using the `perltex.pl` wrapper script.

### 2.1 Defining and redefining Perl macros

`\perlnewcommand`
`\perlrenewcommand`
`\perlnewenvironment`
`\perlrenewenvironment`
`\perldo`

`perltex.sty` defines five macros: `\perlnewcommand`, `\perlrenewcommand`, `\perlnewenvironment`, `\perlrenewenvironment`, and `\perldo`. The first four of these behave exactly like their LaTeX $2_\varepsilon$ counterparts—`\newcommand`, `\renewcommand`, `\newenvironment`, and `\renewenvironment`—except that the macro body consists of Perl code that dynamically generates LaTeX code. `perltex.sty` even includes support for optional arguments and the starred forms of its commands (i.e. `\perlnewcommand*`, `\perlrenewcommand*`, `\perlnewenvironment*`, and `\perlrenewenvironment*`). `\perldo` immediately executes a block of Perl code without (re)defining any macros or environments.

A PerlTeX-defined macro or environments is converted to a Perl subroutine named after the macro/environment but beginning with "`latex_`". For example, a PerlTeX-defined LaTeX macro called `\myMacro` internally produces a Perl subroutine called `latex_myMacro`. Macro arguments are converted to subroutine arguments. A LaTeX macro's `#1` argument is referred to as `$_[0]` in Perl; `#2` is referred to as `$_[1]`; and so forth.

Any valid Perl code can be used in the body of a macro. However, PerlTEX executes the Perl code within a secure sandbox. This means that potentially harmful Perl operations, such as `unlink`, `rmdir`, and `system` will result in a run-time error. (It is possible to disable the safety checks, however, as is explained in Section 2.2.) Having a secure sandbox implies that it is safe to build PerlTEX documents written by other people without worrying about what they may do to your computer system.

A single sandbox is used for the entire `latex` run. This means that multiple macros defined by `\perlnewcommand` can invoke each other. It also means that global variables persist across macro calls:

```
\perlnewcommand{\setX}[1]{$x = $_[0]; return ""}
\perlnewcommand{\getX}{'$x$ was set to ' . $x . '.'}
\setX{123}
\getX
\setX{456}
\getX
\perldo{$x = 789}
\getX
```

$$\Downarrow$$

*x* was set to 123. *x* was set to 456. *x* was set to 789.

Macro arguments are expanded by LATEX before being passed to Perl. Consider the following macro definition, which wraps its argument within `\begin{verbatim*}`...`\end{verbatim*}`:

```
\perlnewcommand{\verbit}[1]{
  "\\begin{verbatim*}\n$_[0]\n\\end{verbatim*}\n"
}
```

An invocation of "`\verbit{\TeX}`" would therefore typeset the *expansion* of "`\TeX`", namely "`T\kern -.1667em\lower .5ex\hbox {E}\kern -.125emX\spacefactor \@m`", which might be a bit unexpected. The solution is to use `\noexpand`: `\verbit{\noexpand\TeX}` ⇒ `\TeX`. "Robust" macros as well as `\begin` and `\end` are implicitly preceded by `\noexpand`.

## 2.2 Invoking `perltex.pl`

The following pages reproduce the `perltex.pl` program documentation. Key parts of the documentation are excerpted when `perltex.pl` is invoked with the `--help` option. The various Perl `pod2`⟨*something*⟩ tools can be used to generate the complete program documentation in a variety of formats such as LATEX, HTML, plain text, or Unix man-page format. For example, the following command is the recommended way to produce a Unix man page from `perltex.pl`:

```
pod2man --center=" " --release=" " perltex.pl > perltex.1
```

## NAME

perltex — enable LaTeX macros to be defined in terms of Perl code

## SYNOPSIS

perltex [**--help**] [**--latex**=*program*] [**--**[**no**]**safe**] [**--permit**=*feature*] [*latex options*]

## DESCRIPTION

LaTeX — through the underlying TeX typesetting system — produces beautifully typeset documents but has a macro language that is difficult to program. In particular, support for complex string manipulation is largely lacking. Perl is a popular general-purpose programming language whose forte is string manipulation. However, it has no typesetting capabilities whatsoever.

Clearly, Perl's programmability could complement LaTeX's typesetting strengths. **perltex** is the tool that enables a symbiosis between the two systems. All a user needs to do is compile a LaTeX document using **perltex** instead of **latex**. (**perltex** is actually a wrapper for **latex**, so no **latex** functionality is lost.) If the document includes a `\usepackage{perltex}` in its preamble, then `\perlnewcommand` and `\perlrenewcommand` macros will be made available. These behave just like LaTeX's `\newcommand` and `\renewcommand` except that the macro body contains Perl code instead of LaTeX code.

## OPTIONS

**perltex** accepts the following command-line options:

**--help**
: Display basic usage information.

**--latex=*program***
: Specify a program to use instead of **latex**. For example, `--latex=pdflatex` would typeset the given document using **pdflatex** instead of ordinary **latex**.

**--[no]safe**
: Enable or disable sandboxing. With the default of `--safe`, **perltex** executes the code from a `\perlnewcommand` or `\perlrenewcommand` macro within a protected environment that prohibits "unsafe" operations such as accessing files or executing external programs. Specifying `--nosafe` gives the LaTeX document *carte blanche* to execute any arbitrary Perl code, including that which can harm the user's files. See the *Safe* manpage for more information.

**--permit=*feature***
: Permit particular Perl operations to be performed. The `--permit` option,

which can be specified more than once on the command line, enables finer-grained control over the **perltex** sandbox. See the *Opcode* manpage for more information.

These options are then followed by whatever options are normally passed to **latex** (or whatever program was specified with `--latex`), including, for instance, the name of the *.tex* file to compile.

## EXAMPLES

In its simplest form, **perltex** is run just like **latex**:

```
perltex myfile.tex
```

To use **pdflatex** instead of regular **latex**, use the `--latex` option:

```
perltex --latex=pdflatex myfile.tex
```

If LaTeX gives a "`trapped by operation mask`" error and you trust the *.tex* file you're trying to compile not to execute malicious Perl code (e.g., because you wrote it yourself), you can disable **perltex**'s safety mechansisms with `--nosafe`:

```
perltex --nosafe myfile.tex
```

The following command gives documents only **perltex**'s default permissions (`:browse`) plus the ability to open files and invoke the `time` command:

```
perltex --permit=:browse --permit=:filesys_open
  --permit=time myfile.tex
```

## ENVIRONMENT

**perltex** honors the following environment variables:

**PERLTEX**

Specify the filename of the LaTeX compiler. The LaTeX compiler defaults to "`latex`". The `PERLTEX` environment variable overrides this default, and the `--latex` command-line option (see the `OPTIONS` entry elsewhere in this document) overrides that.

## FILES

While compiling *jobname.tex*, **perltex** makes use of the following files:

***jobname.lgpl***

log file written by Perl; helpful for debugging Perl macros

***jobname.topl***
information sent from LaTeX to Perl

***jobname.frpl***
information sent from Perl to LaTeX

***jobname.tfpl***
"flag" file whose existence indicates that *jobname.topl* contains valid data

***jobname.ffpl***
"flag" file whose existence indicates that *jobname.frpl* contains valid data

***jobname.dfpl***
"flag" file whose existence indicates that *jobname.ffpl* has been deleted

## NOTES

**perltex**'s sandbox defaults to what the *Opcode* manpage calls "`:browse`".

## SEE ALSO

*latex*(1), *pdflatex*(1), *perl*(1), *Safe*(3pm), *Opcode*(3pm)

## AUTHOR

Scott Pakin, *scott+pt@pakin.org*

## 3 Implementation

Users interested only in *using* PerlTEX can skip Section 3, which presents the complete PerlTEX source code. This section should be of interest primarily to those who wish to extend PerlTEX or modify it to use a language other than Perl.

Section 3 is split into two main parts. Section 3.1 presents the source code for `perltex.sty`, the LATEX side of PerlTEX, and Section 3.2 presents the source code for `perltex.pl`, the Perl side of PerlTEX. In toto, PerlTEX consists of a relatively small amount of code. `perltex.sty` is only 224 lines of LATEX and `perltex.pl` is only 230 lines of Perl. `perltex.pl` is fairly straightforward Perl code and shouldn't be too difficult to understand by anyone comfortable with Perl programming. `perltex.sty`, in contrast, contains a bit of LATEX trickery and is probably impenetrable to anyone who hasn't already tried his hand at LATEX programming. Fortunately for the reader, the code is profusely commented so the aspiring LATEX guru may yet learn something from it.

After documenting the `perltex.sty` and `perltex.pl` source code, a few suggestions are provided for porting PerlTEX to use a backend language other than Perl (Section 3.3).

### 3.1 `perltex.sty`

Although I've written a number of LATEX packages, `perltex.sty` was the most challenging to date. The key things I needed to learn how to do include the following:

1. storing brace-matched—but otherwise not valid LATEX—code for later use
2. iterating over a macro's arguments

Storing non-LATEX code in a variable involves beginning a group in an argumentless macro, fiddling with category codes, using `\afterassignment` to specify a continuation function, and storing the subsequent brace-delimited tokens in the input stream into a token register. The continuation function, which also takes no arguments, ends the group begun in the first function and proceeds using the correctly `\catcode`d token register. This technique appears in `\plmac@haveargs` and `\plmac@havecode` and in a simpler form (i.e., without the need for storing the argument) in `\plmac@write@perl` and `\plmac@write@perl@i`.

Iterating over a macro's arguments is hindered by TEX's requirement that "`#`" be followed by a number or another "`#`". The technique I discovered (which is used by the Texinfo source code) is first to `\let` a variable be `\relax`, thereby making it unexpandable, then to define a macro that uses that variable followed by a loop variable, and finally to expand the loop variable and `\let` the `\relax`ed variable be "`#`" right before invoking the macro. This technique appears in `\plmac@havecode`.

I hope you find reading the `perltex.sty` source code instructive. Writing it certainly was.

### 3.1.1 Package initialization

PerlTEX defines six macros that are used for communication between Perl and LATEX. `\plmac@tag` is a string of characters that should never occur within one of the user's macro names, macro arguments, or macro bodies. `perltex.pl` therefore defines `\plmac@tag` as a long string of random uppercase letters. `\plmac@tofile` is the name of a file used for communication from LATEX to Perl. `\plmac@fromfile` is the name of a file used for communication from Perl to LATEX. `\plmac@toflag` signals that `\plmac@tofile` can be read safely. `\plmac@fromflag` signals that `\plmac@fromfile` can be read safely. `\plmac@doneflag` signals that `\plmac@fromflag` has been deleted. Table 1 lists all of these variables along with the value assigned to each by `perltex.pl`.

Table 1: Variables used for communication between Perl and LATEX

| Variable | Purpose | `perltex.pl` assignment |
|---|---|---|
| `\plmac@tag` | `\plmac@tofile` field separator | (20 random letters) |
| `\plmac@tofile` | LATEX → Perl communication | `\jobname.topl` |
| `\plmac@fromfile` | Perl → LATEX communication | `\jobname.frpl` |
| `\plmac@toflag` | `\plmac@tofile` synchronization | `\jobname.tfpl` |
| `\plmac@fromflag` | `\plmac@fromfile` synchronization | `\jobname.ffpl` |
| `\plmac@doneflag` | `\plmac@fromflag` synchronization | `\jobname.dfpl` |

`\ifplmac@have@perltex`
`\plmac@have@perltextrue`
`\plmac@have@perltexfalse`

The following block of code checks the existence of each of the variables listed in Table 1. If any variable is not defined, `perltex.sty` gives an error message and—as we shall see on page 21—defines dummy versions of `\perl[re]newcommand` and `\perl[re]newenvironment`.

```
\newif\ifplmac@have@perltex
\plmac@have@perltextrue
\@ifundefined{plmac@tag}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@tofile}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@fromfile}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@toflag}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@fromflag}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@doneflag}{\plmac@have@perltexfalse}{}
\ifplmac@have@perltex
\else
  \PackageError{perltex}{Document must be compiled using perltex}
    {Instead of compiling your document directly with latex, you need
     to\MessageBreak use the perltex script.  \space perltex sets up
     a variety of macros needed by\MessageBreak the perltex
     package as well as a listener process needed for\MessageBreak
     communication between LaTeX and Perl.}
\fi
```

### 3.1.2 Defining Perl macros

PerlTEX defines five macros intended to be called by the author. Section 3.1.2 details the implementation of two of them: `\perlnewcommand` and `\perlrenewcommand`. (Section 3.1.3 details the implementation of the next two, `\perlnewenvironment` and `\perlrenewenvironment`; and, Section 3.1.4 details the implementation of the final macro, `\perldo`.) The goal is for these two macros to behave *exactly* like `\newcommand` and `\renewcommand`, respectively, except that the author macros they in turn define have Perl bodies instead of LATEX bodies.

The sequence of the operations defined in this section is as follows:

1. The user invokes `\perl`[re]`newcommand`, which stores `\`[re]`newcommand` in `\plmac@command`. The `\perl`[re]`newcommand` macro then invokes `\plmac@newcommand@i` with a first argument of "`*`" for `\perl`[re]`newcommand*` or "`!`" for ordinary `\perl`[re]`newcommand`.

2. `\plmac@newcommand@i` defines `\plmac@starchar` as "`*`" if it was passed a "`*`" or ⟨*empty*⟩ if it was passed a "`!`". It then stores the name of the user's macro in `\plmac@macname`, a `\write`able version of the name in `\plmac@cleaned@macname`, and the macro's previous definition (needed by `\perlrenewcommand`) in `\plmac@oldbody`. Finally, `\plmac@newcommand@i` invokes `\plmac@newcommand@ii`.

3. `\plmac@newcommand@ii` stores the number of arguments to the user's macro (which may be zero) in `\plmac@numargs`. It then invokes `\plmac@newcommand@iii@opt` if the first argument is supposed to be optional or `\plmac@newcommand@iii@no@opt` if all arguments are supposed to be required.

4. `\plmac@newcommand@iii@opt` defines `\plmac@defarg` as the default value of the optional argument. `\plmac@newcommand@iii@no@opt` defines it as ⟨*empty*⟩. Both functions then call `\plmac@haveargs`.

5. `\plmac@haveargs` stores the user's macro body (written in Perl) verbatim in `\plmac@perlcode`. `\plmac@haveargs` then invokes `\plmac@havecode`.

6. By the time `\plmac@havecode` is invoked all of the information needed to define the user's macro is available. Before defining a LATEX macro, however, `\plmac@havecode` invokes `\plmac@write@perl` to tell `perltex.pl` to define a Perl subroutine with a name based on `\plmac@cleaned@macname` and the code contained in `\plmac@perlcode`. Figure 1 illustrates the data that `\plmac@write@perl` passes to `perltex.pl`.

7. `\plmac@havecode` invokes `\newcommand` or `\renewcommand`, as appropriate, defining the user's macro as a call to `\plmac@write@perl`. An invocation of the user's LATEX macro causes `\plmac@write@perl` to pass the information shown in Figure 2 to `perltex.pl`.

| |
|---|
| `DEF` |
| `\plmac@tag` |
| `\plmac@cleaned@macname` |
| `\plmac@tag` |
| `\plmac@perlcode` |

Figure 1: Data written to `\plmac@tofile` to define a Perl subroutine

| |
|---|
| `USE` |
| `\plmac@tag` |
| `\plmac@cleaned@macname` |
| `\plmac@tag` |
| `#1` |
| `\plmac@tag` |
| `#2` |
| `\plmac@tag` |
| `#3` |
| ⋮ |
| #⟨*last*⟩ |

Figure 2: Data written to `\plmac@tofile` to invoke a Perl subroutine

8. Whenever `\plmac@write@perl` is invoked it writes its argument verbatim to `\plmac@tofile`; `perltex.pl` evaluates the code and writes `\plmac@fromfile`; finally, `\plmac@write@perl` `\input`s `\plmac@fromfile`.

An example might help distinguish the myriad macros used internally by `perltex.sty`. Consider the following call made by the user's document:

```
\perlnewcommand*{\example}[3][frobozz]{join("---", @_)}
```

Table 2 shows how `perltex.sty` parses that command into its constituent components and which components are bound to which `perltex.sty` macros.

Table 2: Macro assignments corresponding to an sample `\perlnewcommand*`

| Macro | Sample definition | |
|---|---|---|
| `\plmac@command` | `\newcommand` | |
| `\plmac@starchar` | `*` | |
| `\plmac@macname` | `\example` | |
| `\plmac@cleaned@macname` | `\example` | (catcode 11) |
| `\plmac@oldbody` | `\relax` | (presumably) |
| `\plmac@numargs` | `3` | |
| `\plmac@defarg` | `frobozz` | |
| `\plmac@perlcode` | `join("---", @_)` | (catcode 11) |

`\perlnewcommand` `\perlrenewcommand` `\plmac@command` `\plmac@next`

`\perlnewcommand` and `\perlrenewcommand` are the first two commands exported to the user by `perltex.sty`. `\perlnewcommand` is analogous to `\newcommand` except that the macro body consists of Perl code instead of LaTeX code. Likewise, `\perlrenewcommand` is analogous to `\renewcommand` except that the macro body consists of Perl code instead of LaTeX code. `\perlnewcommand` and `\perlrenewcommand` merely define `\plmac@command` and `\plmac@next` and invoke `\plmac@newcommand@i`.

```
\def\perlnewcommand{%
  \let\plmac@command=\newcommand
  \let\plmac@next=\relax
  \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
}
```

```
\def\perlrenewcommand{%
  \let\plmac@next=\relax
  \let\plmac@command=\renewcommand
  \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
}
```

`\plmac@newcommand@i` `\plmac@starchar` `\plmac@macname` `\plmac@oldbody` `\plmac@cleaned@macname`

If the user invoked `\perl`[re]`newcommand*` then `\plmac@newcommand@i` is passed a "`*`" and, in turn, defines `\plmac@starchar` as "`*`". If the user invoked `\perl`[re]`newcommand` (no "`*`") then `\plmac@newcommand@i` is passed a "`!`" and, in turn, defines `\plmac@starchar` as ⟨*empty*⟩. In either case, `\plmac@newcommand@i` defines `\plmac@macname` as the name of the user's macro, `\plmac@cleaned@macname` as a `\write`able (i.e., category code 11) version of `\plmac@macname`, and `\plmac@oldbody` and the previous definition of the user's macro. (`\plmac@oldbody` is needed by `\perlrenewcommand`.) It then invokes `\plmac@newcommand@ii`.

```
\def\plmac@newcommand@i#1#2{%
  \ifx#1*%
    \def\plmac@starchar{*}%
  \else
    \def\plmac@starchar{}%
  \fi
  \def\plmac@macname{#2}%
  \let\plmac@oldbody=#2\relax
  \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
    \expandafter\string\plmac@macname}%
  \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
}
```

`\plmac@newcommand@ii` `\plmac@numargs`

`\plmac@newcommand@i` invokes `\plmac@newcommand@ii` with the number of arguments to the user's macro in brackets. `\plmac@newcommand@ii` stores that number in `\plmac@numargs` and invokes `\plmac@newcommand@iii@opt` if the first argument is to be optional or `\plmac@newcommand@iii@no@opt` if all arguments are to be mandatory.

```
\def\plmac@newcommand@ii[#1]{%
  \def\plmac@numargs{#1}%
  \@ifnextchar[{\plmac@newcommand@iii@opt}
               {\plmac@newcommand@iii@no@opt}%]
}
```

\plmac@newcommand@iii@opt
\plmac@newcommand@iii@no@opt
\plmac@defarg

Only one of these two macros is executed per invocation of `\perl[re]newcommand`, depending on whether or not the first argument of the user's macro is an optional argument. `\plmac@newcommand@iii@opt` is invoked if the argument is optional. It defines `\plmac@defarg` to the default value of the optional argument. `\plmac@newcommand@iii@no@opt` is invoked if all arguments are mandatory. It defines `\plmac@defarg` as `\relax`. Both `\plmac@newcommand@iii@opt` and `\plmac@newcommand@iii@no@opt` then invoke `\plmac@haveargs`.

```
\def\plmac@newcommand@iii@opt[#1]{%
  \def\plmac@defarg{#1}%
  \plmac@haveargs
}
```

```
\def\plmac@newcommand@iii@no@opt{%
  \let\plmac@defarg=\relax
  \plmac@haveargs
}
```

\plmac@perlcode
\plmac@haveargs

Now things start to get tricky. We have all of the arguments we need to define the user's command so all that's left is to grab the macro body. But there's a catch: Valid Perl code is unlikely to be valid LaTeX code. We therefore have to read the macro body in a `\verb`-like mode. Furthermore, we actually need to *store* the macro body in a variable, as we don't need it right away.

The approach we take in `\plmac@haveargs` is as follows. First, we give all "special" characters category code 12 ("other"). We then indicate that the carriage return character (control-M) marks the end of a line and that curly braces retain their normal meaning. With the aforementioned category-code definitions, we now have to store the next curly-brace-delimited fragment of text, end the current group to reset all category codes to their previous value, and continue processing the user's macro definition. How do we do that? The answer is to assign the upcoming text fragment to a token register (`\plmac@perlcode`) while an `\afterassignment` is in effect. The `\afterassignment` causes control to transfer to `\plmac@havecode` right after `\plmac@perlcode` receives the macro body with all of the "special" characters made impotent.

```
\newtoks\plmac@perlcode
```

```
\def\plmac@haveargs{%
  \begingroup
    \let\do\@makeother\dospecials
    \catcode`\^^M=\active
    \newlinechar`\^^M
    \endlinechar=`\^^M
    \catcode`\{=1
    \catcode`\}=2
    \afterassignment\plmac@havecode
    \global\plmac@perlcode
}
```

Control is transfered to `\plmac@havecode` from `\plmac@haveargs` right after the user's macro body is assigned to `\plmac@perlcode`. We now have everything we need to define the user's macro. The goal is to define it as "`\plmac@write@perl{`⟨*contents of Figure 2*⟩`}`". This is easier said than done because the number of arguments in the user's macro is not known statically, yet we need to iterate over however many arguments there are. Because of this complexity, we will explain `\plmac@perlcode` piece-by-piece.

`\plmac@sep` Define a character to separate each of the items presented in Figures 1 and 2. Perl will need to strip this off each argument. For convenience in porting to languages with less powerful string manipulation than Perl's, we define `\plmac@sep` as a carriage-return character of category code 11 ("letter").

```
{\catcode`\^^M=11\gdef\plmac@sep{^^M}}
```

`\plmac@argnum` Define a loop variable that will iterate from `1` to the number of arguments in the user's function, i.e., `\plmac@numargs`.

```
\newcount\plmac@argnum
```

`\plmac@havecode` Now comes the final piece of what started as a call to `\perl`[re]`newcommand`. First, to reset all category codes back to normal, `\plmac@havecode` ends the group that was begun in `\plmac@haveargs`.

```
\def\plmac@havecode{%
  \endgroup
```

`\plmac@define@sub` We invoke `\plmac@write@perl` to define a Perl subroutine named after `\plmac@cleaned@macname`. `\plmac@define@sub` sends Perl the information shown in Figure 1 on page 12.

```
  \edef\plmac@define@sub{%
    \noexpand\plmac@write@perl{DEF\plmac@sep
      \plmac@tag\plmac@sep
      \plmac@cleaned@macname\plmac@sep
      \plmac@tag\plmac@sep
      \the\plmac@perlcode
    }%
  }%
  \plmac@define@sub
```

`\plmac@body` The rest of `\plmac@havecode` is preparation for defining the user's macro. (LaTeX $2_\varepsilon$'s `\newcommand` or `\renewcommand` will do the actual work, though.) `\plmac@body` will eventually contain the complete (LaTeX) body of the user's macro. Here, we initialize it to the first three items listed in Figure 2 on page 12 (with intervening `\plmac@sep`s).

```
  \edef\plmac@body{%
    USE\plmac@sep
    \plmac@tag\plmac@sep
    \plmac@cleaned@macname
  }%
```

`\plmac@hash` Now, for each argument `#1`, `#2`, …, `#\plmac@numargs` we append a `\plmac@tag` plus the argument to `\plmac@body` (as always, with a `\plmac@sep` after each item). This requires more trickery, as TeX requires a macro-parameter character (“`#`”) to be followed by a literal number, not a variable. The approach we take, which I first discovered in the Texinfo source code (although it’s used by LaTeX and probably other TeX-based systems as well), is to `\let`-bind `\plmac@hash` to `\relax`. This makes `\plmac@hash` unexpandable, and because it’s not a “`#`”, TeX doesn’t complain. After `\plmac@body` has been extended to include `\plmac@hash1`, `\plmac@hash2`, …, `\plmac@hash\plmac@numargs`, we then `\let`-bind `\plmac@hash` to `##`, which TeX lets us do because we’re within a macro definition (`\plmac@havecode`). `\plmac@body` will then contain `#1`, `#2`, …, `#\plmac@numargs`, as desired.

```
\let\plmac@hash=\relax
\plmac@argnum=\@ne
\loop
  \ifnum\plmac@numargs<\plmac@argnum
  \else
    \edef\plmac@body{%
      \plmac@body\plmac@sep\plmac@tag\plmac@sep
      \plmac@hash\plmac@hash\number\plmac@argnum}%
    \advance\plmac@argnum by \@ne
\repeat
\let\plmac@hash=##%
```

`\plmac@define@command` We’re ready to execute a `\[re]newcommand`. Because we need to expand many of our variables, we `\edef \plmac@define@command` to the appropriate `\[re]newcommand` call, which we will soon execute. The user’s macro must first be `\let`-bound to `\relax` to prevent it from expanding. Then, we handle two cases: either all arguments are mandatory (and `\plmac@defarg` is `\relax`) or the user’s macro has an optional argument (with default value `\plmac@defarg`).

```
\expandafter\let\plmac@macname=\relax
\ifx\plmac@defarg\relax
  \edef\plmac@define@command{%
    \noexpand\plmac@command\plmac@starchar{\plmac@macname}%
    [\plmac@numargs]{%
      \noexpand\plmac@write@perl{\plmac@body}%
    }%
}%
\else
  \edef\plmac@define@command{%
    \noexpand\plmac@command\plmac@starchar{\plmac@macname}%
    [\plmac@numargs][\plmac@defarg]{%
      \noexpand\plmac@write@perl{\plmac@body}%
    }%
}%
\fi
```

The final steps are to restore the previous definition of the user’s macro—we had set it to `\relax` above to make the name unexpandable—then redefine it by invoking `\plmac@define@command`. Why do we need to restore the previous definition if we're just going to redefine it? Because `\newcommand` needs to produce an error if the macro was previously defined and `\renewcommand` needs to produce an error if the macro was *not* previously defined.

`\plmac@havecode` concludes by invoking `\plmac@next`, which is a no-op for `\perlnewcommand` and `\perlrenewcommand` but processes the end-environment code for `\perlnewenvironment` and `\perlrenewenvironment`.

```
  \expandafter\let\plmac@macname=\plmac@oldbody
  \plmac@define@command
  \plmac@next
}
```

### 3.1.3 Defining Perl environments

Section 3.1.2 detailed the implementation of `\perlnewcommand` and `\perlrenewcommand`. Section 3.1.3 does likewise for `\perlnewenvironment` and `\perlrenewenvironment`, which are the Perl-bodied analogues of `\newenvironment` and `\renewenvironment`. This section is significantly shorter than the previous because `\perlnewenvironment` and `\perlrenewenvironment` are largely built atop the macros already defined in Section 3.1.2.

`\perlnewenvironment`
`\perlrenewenvironment`
`\plmac@command`
`\plmac@next`

`\perlnewenvironment` and `\perlrenewenvironment` are the remaining two commands exported to the user by `perltex.sty`. `\perlnewenvironment` is analogous to `\newenvironment` except that the macro body consists of Perl code instead of LaTeX code. Likewise, `\perlrenewenvironment` is analogous to `\renewenvironment` except that the macro body consists of Perl code instead of LaTeX code. `\perlnewenvironment` and `\perlrenewenvironment` merely define `\plmac@command` and `\plmac@next` and invoke `\plmac@newenvironment@i`.

The significance of `\plmac@next` (which was let-bound to `\relax` for `\perl[re]newcommand` but is let-bound to `\plmac@end@environment` here) is that a LaTeX environment definition is really two macro definitions: `\`⟨*name*⟩ and `\end`⟨*name*⟩. Because we want to reuse as much code as possible the idea is to define the "begin" code as one macro, then inject—by way of `plmac@next`—a call to `\plmac@end@environment`, which defines the "end" code as a second macro.

```
\def\perlnewenvironment{%
  \let\plmac@command=\newcommand
  \let\plmac@next=\plmac@end@environment
  \@ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
}
```

```
\def\perlrenewenvironment{%
  \let\plmac@command=\renewcommand
  \let\plmac@next=\plmac@end@environment
  \@ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
}
```

`\plmac@newenvironment@i`
`\plmac@starchar`
`\plmac@envname`
`\plmac@macname`
`\plmac@oldbody`
`\plmac@cleaned@macname`

The `\plmac@newenvironment@i` macro is analogous to `\plmac@newcommand@i`; see the description of `\plmac@newcommand@i` on page 13 to understand the basic structure. The primary difference is that the environment name (#2) is just text, not a control sequence. We store this text in \plmac@envname to facilitate generating the names of the two macros that constitute an environment definition. Note that there is no \plmac@newenvironment@ii; control passes instead to \plmac@newcommand@ii.

```
\def\plmac@newenvironment@i#1#2{%
  \ifx#1*%
    \def\plmac@starchar{*}%
  \else
    \def\plmac@starchar{}%
  \fi
  \def\plmac@envname{#2}%
  \expandafter\def\expandafter\plmac@macname\expandafter{\csname#2\endcsname}%
  \expandafter\let\expandafter\plmac@oldbody\plmac@macname\relax
  \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
    \expandafter\string\plmac@macname}%
  \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
}
```

\plmac@end@environment
\plmac@next
\plmac@macname
\plmac@oldbody
\plmac@cleaned@macname

Recall that an environment definition is a shortcut for two macro definitions: \⟨*name*⟩ and \end⟨*name*⟩ (where ⟨*name*⟩ was stored in \plmac@envname by \plmac@newenvironment@i). After defining \⟨*name*⟩, \plmac@havecode transfers control to \plmac@end@environment because \plmac@next was let-bound to \plmac@end@environment in \perl[re]newenvironment.

\plmac@end@environment's purpose is to define \end⟨*name*⟩. This is a little tricky, however, because LaTeX's \[re]newcommand refuses to (re)define a macro whose name begins with "end". The solution that \plmac@end@environment takes is first to define a \plmac@end@macro macro then (in plmac@next) let-bind \end⟨*name*⟩ to it. Other than that, \plmac@end@environment is a combined and simplified version of \perlnewenvironment, \perlrenewenvironment, and \plmac@newenvironment@i.

```
\def\plmac@end@environment{%
  \expandafter\def\expandafter\plmac@next\expandafter{\expandafter
    \let\csname end\plmac@envname\endcsname=\plmac@end@macro
    \let\plmac@next=\relax
  }%
  \def\plmac@macname{\plmac@end@macro}%
  \expandafter\let\expandafter\plmac@oldbody\csname end\plmac@envname\endcsname
  \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
    \expandafter\string\plmac@macname}%
  \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
}
```

### 3.1.4 Executing top-level Perl code

The macros defined in Sections 3.1.2 and 3.1.3 enable an author to inject subroutines into the Perl sandbox. The final PerlTeX macro, \perldo, instructs the Perl sandbox to execute a block of code outside of all subroutines. `\perldo`'s implementation is much simpler than that of the other author macros because `\perldo` does not have to process subroutine arguments. Figure 3 illustrates the data that gets written to `plmac@tofile` (indirectly) by `\perldo`.

| |
|---|
| `RUN` |
| `\plmac@tag` |
| *Ignored* |
| `\plmac@tag` |
| `\plmac@perlcode` |

Figure 3: Data written to `\plmac@tofile` to execute Perl code

`\perldo` Execute a block of Perl code and pass the result to LaTeX for further processing. This code is nearly identical to that of Section 3.1.2's `\plmac@haveargs` but ends by invoking `\plmac@have@run@code` instead of `\plmac@havecode`.

```
\def\perldo{%
  \begingroup
    \let\do\@makeother\dospecials
    \catcode`\^^M=\active
    \newlinechar`\^^M
    \endlinechar=`\^^M
    \catcode`\{=1
    \catcode`\}=2
    \afterassignment\plmac@have@run@code
    \global\plmac@perlcode
}
```

`\plmac@have@run@code`
`\plmac@run@code` Pass a block of code to Perl to execute. `\plmac@have@run@code` is identical to `\plmac@havecode` but specifies the `RUN` tag instead of the `DEF` tag.

```
\def\plmac@have@run@code{%
  \endgroup
  \edef\plmac@run@code{%
    \noexpand\plmac@write@perl{RUN\plmac@sep
      \plmac@tag\plmac@sep
      N/A\plmac@sep
      \plmac@tag\plmac@sep
      \the\plmac@perlcode
    }%
  }%
  \plmac@run@code
}
```

### 3.1.5 Communication between LaTeX and Perl

As shown in the previous section, when a document invokes `\perl`[re]`newcommand` to define a macro, `perltex.sty` defines the macro in terms of a call to `\plmac@write@perl`. In this section, we learn how `\plmac@write@perl` operates.

At the highest level, LaTeX-to-Perl communication is performed via the filesystem. In essence, LaTeX writes a file (`\plmac@tofile`) corresponding to the information in either Figure 1 or Figure 2; Perl reads the file, executes the code within it, and writes a `.tex` file (`\plmac@fromfile`); and, finally, LaTeX reads and executes the new `.tex` file. However, the actual communication protocol is a bit more involved than that. The problem is that Perl needs to know when LaTeX has finished writing Perl code and LaTeX needs to know when Perl has finished writing LaTeX code. The solution involves introducing three extra files—`\plmac@toflag`, `\plmac@fromflag`, and `\plmac@doneflag`—which are used exclusively for LaTeX-to-Perl synchronization.

There's a catch: Although Perl can create and delete files, LaTeX can only create them. Even worse, LaTeX (more specifically, teTeX, which is the TeX distribution under which I developed PerlTeX) cannot reliably poll for a file's *non*existence; if a file is deleted in the middle of an `\immediate\openin`, `latex` aborts with an error message. These restrictions led to the regrettably convoluted protocol illustrated in Figure 4. In the figure, "Touch" means "create a zero-length file"; "Await" means "wait until the file exists"; and, "Read", "Write", and "Delete" are defined as expected. Assuming the filesystem performs these operations in a sequentially consistent order (not necessarily guaranteed on all filesystems, unfortunately), PerlTeX should behave as expected.

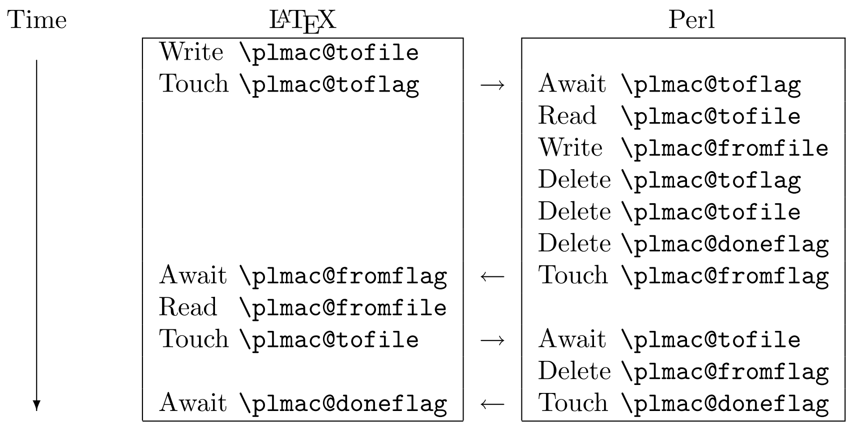

Figure 4: LaTeX-to-Perl communication protocol

`\plmac@await@existence` `\ifplmac@file@exists` `\plmac@file@existstrue` `\plmac@file@existsfalse`

The purpose of the `\plmac@await@existence` macro is to repeatedly check the existence of a given file until the file actually exists. For convenience, we use LaTeX 2$_\varepsilon$'s `\IfFileExists` macro to check the file and invoke `\plmac@file@existstrue` or `\plmac@file@existsfalse`, as appropriate.

```
\newif\ifplmac@file@exists

\newcommand{\plmac@await@existence}[1]{%
  \loop
    \IfFileExists{#1}%
```

```
              {\plmac@file@existstrue}%
              {\plmac@file@existsfalse}%
    \ifplmac@file@exists
    \else
  \repeat
}
```

`\plmac@outfile` We define a file handle for `\plmac@write@perl@i` to use to create and write `\plmac@tofile` and `\plmac@toflag`.

```
\newwrite\plmac@outfile
```

`\plmac@write@perl` `\plmac@write@perl` begins the LaTeX-to-Perl data exchange, following the protocol illustrated in Figure 4. `\plmac@write@perl` prepares for the next piece of text in the input stream to be read with "special" characters marked as category code 12 ("other"). This prevents LaTeX from complaining if the Perl code contains invalid LaTeX (which it usually will). `\plmac@write@perl` ends by passing control to `\plmac@write@perl@i`, which performs the bulk of the work.

```
\newcommand{\plmac@write@perl}{%
  \begingroup
    \let\do\@makeother\dospecials
    \catcode`\^^M=\active
    \newlinechar`\^^M
    \endlinechar=`\^^M
    \catcode`\{=1
    \catcode`\}=2
    \plmac@write@perl@i
}
```

`\plmac@write@perl@i` When `\plmac@write@perl@i` begins executing, the category codes are set up so that the macro's argument will be evaluated "verbatim" except for the part consisting of the LaTeX code passed in by the author, which is partially expanded. Thus, everything is in place for `\plmac@write@perl@i` to send its argument to Perl and read back the (LaTeX) result.

Because all of `perltex.sty`'s protocol processing is encapsulated within `\plmac@write@perl@i`, this is the only macro that strictly requires `perltex.pl`. Consequently, we wrap the entire macro definition within a check for `perltex.pl`.

```
\ifplmac@have@perltex
  \newcommand{\plmac@write@perl@i}[1]{%
```

The first step is to write argument `#1` to `\plmac@tofile`:

```
      \immediate\openout\plmac@outfile=\plmac@tofile\relax
      \let\protect=\noexpand
      \def\begin{\noexpand\begin}%
      \def\end{\noexpand\end}%
      \immediate\write\plmac@outfile{#1}%
      \immediate\closeout\plmac@outfile
```

(In the future, it might be worth redefining `\def`, `\edef`, `\gdef`, `\xdef`, `\let`, and maybe some other control sequences as "`\noexpand`⟨*control sequence*⟩`\noexpand`" so that `\write` doesn't try to expand an undefined control sequence.)

We're now finished using `#1` so we can end the group begun by `\plmac@write@perl`, thereby resetting each character's category code back to its previous value.

```
    \endgroup
```

Continuing the protocol illustrated in Figure 4, we create a zero-byte `\plmac@toflag` in order to notify `perltex.pl` that it's now safe to read `\plmac@tofile`.

```
    \immediate\openout\plmac@outfile=\plmac@toflag\relax
    \immediate\closeout\plmac@outfile
```

To avoid reading `\plmac@fromfile` before `perltex.pl` has finished writing it we must wait until `perltex.pl` creates `\plmac@fromflag`, which it does only after it has written `\plmac@fromfile`.

```
    \plmac@await@existence\plmac@fromflag
```

At this point, `\plmac@fromfile` should contain valid LaTeX code. However, we defer inputting it until we the very end. Doing so enables recursive and mutually recursive invocations of PerlTeX macros.

Because TeX can't delete files we require an additional LaTeX-to-Perl synchronization step. For convenience, we recycle `\plmac@tofile` as a synchronization file rather than introduce yet another flag file to complement `\plmac@toflag`, `\plmac@fromflag`, and `\plmac@doneflag`.

```
    \immediate\openout\plmac@outfile=\plmac@tofile\relax
    \immediate\closeout\plmac@outfile
    \plmac@await@existence\plmac@doneflag
```

The only thing left to do is to `\input` and evaluate `\plmac@fromfile`, which contains the LaTeX output from the Perl subroutine.

```
    \input\plmac@fromfile\relax
  }
```

The foregoing code represents the "real" definition of `\plmac@write@perl@i`. For the user's convenience, we define a dummy version of `\plmac@write@perl@i` so that a document which utilizes `perltex.sty` can still compile even if not built using `perltex.pl`. All calls to macros defined with `\perl[re]newcommand` and all invocations of environments defined with `\perl[re]newenvironment` are replaced with "$\boxed{\text{PerlTeX}}$". A minor complication is that text can't be inserted before the `\begin{document}`. Hence, we initially define `\plmac@write@perl@i` as a do-nothing macro and redefine it as "`\fbox{Perl\TeX}`" at the `\begin{document}`.

```
\else
  \newcommand{\plmac@write@perl@i}[1]{\endgroup}
  \AtBeginDocument{%
    \renewcommand{\plmac@write@perl@i}[1]{%
```

`\plmac@show@placeholder` There's really no point in outputting a framed "PerlTeX" when a macro is defined *and* when it's used. `\plmac@show@placeholder` checks the first character of the protocol header. If it's "D" (`DEF`), nothing is output. Otherwise, it'll be "U" (`USE`) and "PerlTeX" will be output.

```
      \def\plmac@show@placeholder##1##2\@empty{%
        \ifx##1D\relax
          \endgroup
        \else
          \endgroup
          \fbox{Perl\TeX}%
        \fi
      }%
      \plmac@show@placeholder#1\@empty
    }%
  }
\fi
```

## 3.2 perltex.pl

perltex.pl is a wrapper script for latex (or any other LaTeX compiler). It sets up client-server communication between LaTeX and Perl, with LaTeX as the client and Perl as the server. When a LaTeX document sends a piece of Perl code to perltex.pl (with the help of perltex.sty, as detailed in Section 3.1), perltex.pl executes it within a secure sandbox and transmits the resulting LaTeX code back to the document.

### 3.2.1 Header comments

Because perltex.pl is generated without a DocStrip preamble or postamble we have to manually include the desired text as Perl comments.

```
#! /usr/bin/env perl

###########################################################
# Prepare a LaTeX run for two-way communication with Perl #
# By Scott Pakin <scott+pt@pakin.org>                     #
###########################################################

#-------------------------------------------------------------------
# This is file ‘perltex.pl’,
# generated with the docstrip utility.
#
# The original source files were:
#
# perltex.dtx  (with options: ‘perltex’)
#
# This is a generated file.
#
# Copyright (C) 2006, Scott Pakin <scott+pt@pakin.org>
#
# This file may be distributed and/or modified under the conditions
# of the LaTeX Project Public License, either version 1.3c of this
# license or (at your option) any later version.  The latest
# version of this license is in:
```

```
#
#    http://www.latex-project.org/lppl.txt
#
# and version 1.3c or later is part of all distributions of LaTeX
# version 2006/05/20 or later.
#-----------------------------------------------------------------------

```

### 3.2.2 Top-level code evaluation

In previous versions of `perltex.pl`, the `--nosafe` option created and ran code within a sandbox in which all operations are allowed (via `Opcode::full_opset()`). Unfortunately, certain operations still fail to work within such a sandbox. We therefore define a top-level "non-sandbox", `top_level_eval()`, in which to execute code. `top_level_eval()` merely calls `eval()` on its argument. However, it needs to be declared top-level and before anything else because `eval()` runs in the lexical scope of its caller.

```
sub top_level_eval ($)
{
    return eval $_[0];
}
```

### 3.2.3 Perl modules and pragmas

We use `Safe` and `Opcode` to implement the secure sandbox, `Getopt::Long` and `Pod::Usage` to parse the command line, and various other modules and pragmas for miscellaneous things.

```
use Safe;
use Opcode;
use Getopt::Long;
use Pod::Usage;
use File::Basename;
use POSIX;
use warnings;
use strict;
```

### 3.2.4 Variable declarations

With `use strict` in effect, we need to declare all of our variables. For clarity, we separate our global-variable declarations into variables corresponding to command-line options and other global variables.

#### Variables corresponding to command-line arguments

`$latexprog` `$runsafely` `@permittedops` `$latexprog` is the name of the LaTeX executable (e.g., "`latex`"). If `$runsafely` is `1` (the default), then the user's Perl code runs in a secure sandbox; if it's `0`, then arbitrary Perl code is allowed to run. `@permittedops` is a list of features made available to the user's Perl code. Valid values are described in Perl's `Opcode` manual page. `perltex.pl`'s default is a list containing only `:browse`.

```
my $latexprog;
my $runsafely = 1;
my @permittedops;
```

**Other global variables**

`$progname`
`$jobname`
`@latexcmdline`
`$toperl`
`$fromperl`
`$toflag`
`$fromflag`
`$doneflag`
`$logfile`
`$sandbox`
`$sandbox_eval`
`$latexpid`

`$progname` is the run-time name of the `perltex.pl` program. `$jobname` is the base name of the user's `.tex` file, which defaults to the TEX default of `texput`. `@latexcmdline` is the command line to pass to the LATEX executable. `$toperl` defines the filename used for LATEX→Perl communication. `$fromperl` defines the filename used for Perl→LATEX communication. `$toflag` is the name of a file that will exist only after LATEX creates `$tofile`. `$fromflag` is the name of a file that will exist only after Perl creates `$fromfile`. `$doneflag` is the name of a file that will exist only after Perl deletes `$fromflag`. `$logfile` is the name of a log file to which `perltex.pl` writes verbose execution information. `$sandbox` is a secure sandbox in which to run code that appeared in the LATEX document. `$sandbox_eval` is a subroutine that evalutes a string within `$sandbox` (normally) or outside of all sandboxes (if `--nosafe` is specified). `$latexpid` is the process ID of the `latex` process.

```
my $progname = basename $0;
my $jobname = "texput";
my @latexcmdline;
my $toperl;
my $fromperl;
my $toflag;
my $fromflag;
my $doneflag;
my $logfile;
my $sandbox = new Safe;
my $sandbox_eval;
my $latexpid;
```

### 3.2.5 Command-line conversion

In this section, `perltex.pl` parses its own command line and prepares a command line to pass to `latex`.

**Parsing `perltex.pl`'s command line** We first set `$latexprog` to be the contents of the environment variable `PERLTEX` or the value "`latex`" if `PERLTEX` is not specified. We then use `Getopt::Long` to parse the command line, leaving any parameters we don't recognize in the argument vector (`@ARGV`) because these are presumably `latex` options.

```
$latexprog = $ENV{"PERLTEX"} || "latex";
Getopt::Long::Configure("require_order", "pass_through");
GetOptions("help"     => sub {pod2usage(-verbose => 1)},
           "latex=s"  => \$latexprog,
           "safe!"    => \$runsafely,
           "permit=s" => \@permittedops) || pod2usage(2);
```

### Preparing a LaTeX command line

`$firstcmd` `$option` We start by searching `@ARGV` for the first string that does not start with "-" or "\". This string, which represents a filename, is used to set `$jobname`.

```
@latexcmdline = @ARGV;
my $firstcmd = 0;
for ($firstcmd=0; $firstcmd<=$#latexcmdline; $firstcmd++) {
    my $option = $latexcmdline[$firstcmd];
    next if substr($option, 0, 1) eq "-";
    if (substr ($option, 0, 1) ne "\\") {
        $jobname = basename $option, ".tex" ;
        $latexcmdline[$firstcmd] = "\\input $option";
    }
    last;
}
push @latexcmdline, "" if $#latexcmdline==-1;
```

`$separator` To avoid conflicts with the code and parameters passed to Perl from LaTeX (see Figure 1 on page 12 and Figure 2 on page 12) we define a separator string, `$separator`, containing 20 random uppercase letters.

```
my $separator = "";
foreach (1 .. 20) {
    $separator .= chr(ord("A") + rand(26));
}
```

Now that we have the name of the LaTeX job (`$jobname`) we can assign `$toperl`, `$fromperl`, `$toflag`, `$fromflag`, `$doneflag`, and `$logfile` in terms of `$jobname` plus a suitable extension.

```
$toperl = $jobname . ".topl";
$fromperl = $jobname . ".frpl";
$toflag = $jobname . ".tfpl";
$fromflag = $jobname . ".ffpl";
$doneflag = $jobname . ".dfpl";
$logfile = $jobname . ".lgpl";
```

We now replace the filename of the `.tex` file passed to `perltex.pl` with a `\def`inition of the separator character, `\def`initions of the various files, and the original file with `\input` prepended if necessary.

```
$latexcmdline[$firstcmd] =
    sprintf '\makeatletter' . '\def%s{%s}' x 6 . '\makeatother%s',
    '\plmac@tag', $separator,
    '\plmac@tofile', $toperl,
    '\plmac@fromfile', $fromperl,
    '\plmac@toflag', $toflag,
    '\plmac@fromflag', $fromflag,
    '\plmac@doneflag', $doneflag,
    $latexcmdline[$firstcmd];
```

### 3.2.6 Launching LaTeX

We start by deleting the `$toperl`, `$fromperl`, `$toflag`, `$fromflag`, and `$doneflag` files, in case any of these were left over from a previous (aborted) run. We also create a log file, `$logfile`. As `@latexcmdline` contains the complete command line to pass to `latex` we need only `fork` a new process and have the child process overlay itself with `latex`. `perltex.pl` continues running as the parent.

Note that here and elsewhere in `perltex.pl`, `unlink` is called repeatedly until the file is actually deleted. This works around a race condition that occurs in some filesystems in which file deletions are executed somewhat lazily.

```
foreach my $file ($toperl, $fromperl, $toflag, $fromflag, $doneflag) {
    unlink $file while -e $file;
}
open (LOGFILE, ">$logfile") || die "open(\"$logfile\"): $!\n";
defined ($latexpid = fork) || die "fork: $!\n";
unshift @latexcmdline, $latexprog;
if (!$latexpid) {
    exec {$latexcmdline[0]} @latexcmdline;
    die "exec('@latexcmdline'): $!\n";
}
```

### 3.2.7 Preparing a sandbox

`perltex.pl` uses Perl's `Safe` and `Opcode` modules to declare a secure sandbox (`$sandbox`) in which to run Perl code passed to it from LaTeX. When the sandbox compiles and executes Perl code, it permits only operations that are deemed safe. For example, the Perl code is allowed by default to assign variables, call functions, and execute loops. However, it is not normally allowed to delete files, kill processes, or invoke other programs. If `perltex.pl` is run with the `--nosafe` option we bypass the sandbox entirely and execute Perl code using an ordinary `eval()` statement.

```
if ($runsafely) {
    @permittedops=(":browse") if $#permittedops==-1;
    $sandbox->permit_only (@permittedops);
    $sandbox_eval = sub {$sandbox->reval($_[0])};
}
else {
    $sandbox_eval = \&top_level_eval;
}
```

### 3.2.8 Communicating with LaTeX

The following code constitutes `perltex.pl`'s main loop. Until `latex` exits, the loop repeatedly reads Perl code from LaTeX, evaluates it, and returns the result as per the protocol described in Figure 4 on page 20.

```
while (1) {
```

$awaitexists We define a local subroutine $awaitexists which waits for a given file to exist. If latex exits while $awaitexists is waiting, then perltex.pl cleans up and exits, too.

```
    my $awaitexists = sub {
      while (!-e $_[0]) {
          sleep 0;
          if (waitpid($latexpid, &WNOHANG)==-1) {
              foreach my $file ($toperl, $fromperl, $toflag,
                                $fromflag, $doneflag) {
                  unlink $file while -e $file;
              }
              undef $latexpid;
              exit 0;
          }
      }
    };
```

$entirefile Wait for $toflag to exist. When it does, this implies that $toperl must exist as well. We read the entire contents of $toperl into the $entirefile variable and process it. Figures 1 and 2 illustrate the contents of $toperl.

```
    $awaitexists->($toflag);
    my $entirefile;
    {
        local $/ = undef;
        open (TOPERL, "<$toperl") || die "open($toperl): $!\n";
        $entirefile = <TOPERL>;
        close TOPERL;
    }
```

$optag $macroname @otherstuff We split the contents of $entirefile into an operation tag (either DEF, USE, or RUN), the macro name, and everything else (@otherstuff). If $optag is DEF then @otherstuff will contain the Perl code to define. If $optag is USE then @otherstuff will be a list of subroutine arguments. If $optag is RUN then @otherstuff will be a block of Perl code to run.

```
    my ($optag, $macroname, @otherstuff) =
        map {chomp; $_} split "$separator\n", $entirefile;
```

We clean up the macro name by deleting all leading non-letters, replacing all subsequent non-alphanumerics with “_”, and prepending “latex_” to the macro name.

```
    $macroname =~ s/^[^A-Za-z]+//;
    $macroname =~ s/\W/_/g;
    $macroname = "latex_" . $macroname;
```

If we're calling a subroutine, then we make the arguments more palatable to Perl by single-quoting them and replacing every occurrence of "\" with "\\" and every occurrence of "'" with "\'".

```
    if ($optag eq "USE") {
      foreach (@otherstuff) {
          s/\\/\\\\/g;
          s/\'/\\\'/g;
          $_ = "'$_'";
      }
    }
```

`$perlcode` There are three possible values that can be assigned to `$perlcode`. If `$optag` is `DEF`, then `$perlcode` is made to contain a definition of the user's subroutine, named `$macroname`. If `$optag` is `USE`, then `$perlcode` becomes an invocation of `$macroname` which gets passed all of the macro arguments. Finally, if `$optag` is `RUN`, then `$perlcode` is the unmodified Perl code passed to us from `perltex.sty`. Figure 5 presents an example of how the following code converts a PerlTEX macro definition into a Perl subroutine definition and Figure 6 presents an example of how the following code converts a PerlTEX macro invocation into a Perl subroutine invocation.

```
    my $perlcode;
    if ($optag eq "DEF") {
        $perlcode =
            sprintf "sub %s {%s}\n",
            $macroname, $otherstuff[0];
    }
    elsif ($optag eq "USE") {
        $perlcode = sprintf "%s (%s);\n", $macroname, join(", ", @otherstuff);
    }
    elsif ($optag eq "RUN") {
        $perlcode = $otherstuff[0];
    }
```

LATEX:

```
\perlnewcommand{\mymacro}[2]{%
  sprintf "Isn't $_[0] %s $_[1]?\n",
    $_[0]>=$_[1] ? ">=" : "<"
}
```

$$\Downarrow$$

Perl:

```
sub latex_mymacro {
  sprintf "Isn't $_[0] %s $_[1]?\n",
    $_[0]>=$_[1] ? ">=" : "<"
}
```

Figure 5: Conversion from LATEX to Perl (subroutine definition)

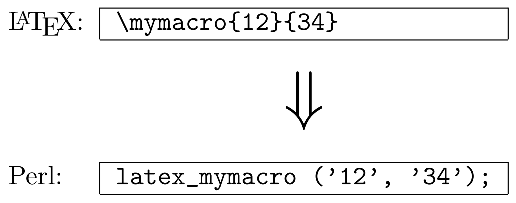

Figure 6: Conversion from LaTeX to Perl (subroutine invocation)

```
    else {
        die "${progname}: Internal error -- unexpected operation tag \"$optag\"\n";
    }
```

Log what we're about to evaluate.

```
    print LOGFILE "#" x 31, " PERL CODE ", "#" x 32, "\n";
    print LOGFILE $perlcode, "\n";
```

`$result` `$msg` We're now ready to execute the user's code using the `$sandbox_eval` function. If a warning occurs we write it as a Perl comment to the log file. If an error occurs (i.e., `$@` is defined) we replace the result (`$result`) with a call to LaTeX 2$_\varepsilon$'s `\PackageError` macro to return a suitable error message. We produce one error message for sandbox policy violations (detected by the error message, `$@`, containing the string "`trapped by`") and a different error message for all other errors caused by executing the user's code. For clarity of reading both warning and error messages, we elide the string "`at (eval` ⟨*number*⟩`) line` ⟨*number*⟩".

```
    undef $_;
    my $result;
    {
        my $warningmsg;
        local $SIG{__WARN__} =
            sub {chomp ($warningmsg=$_[0]); return 0};
        $result = $sandbox_eval->($perlcode);
        if (defined $warningmsg) {
            $warningmsg =~ s/at \(eval \d+\) line \d+\W+//;
            print LOGFILE "# ===> $warningmsg\n\n";
        }
    }
    $result="" if !$result || $optag eq "RUN";
    if ($@) {
        my $msg = $@;
        $msg =~ s/at \(eval \d+\) line \d+\W+//;
        $msg =~ s/\s+/ /;
        $result = "\\PackageError{perltex}{$msg}";
        my @helpstring;
        if ($msg =~ /\btrapped by\b/) {
            @helpstring =
                ("The preceding error message comes from Perl.  Apparently,",
                 "the Perl code you tried to execute attempted to perform an",
                  "‘unsafe’ operation.  If you trust the Perl code (e.g., if",
                  "you wrote it) then you can invoke perltex with the --nosafe",
                  "option to allow arbitrary Perl code to execute.",
                  "Alternatively, you can selectively enable Perl features",
                  "using perltex’s --permit option.  Don’t do this if you don’t",
                  "trust the Perl code, however; malicious Perl code can do a",
                  "world of harm to your computer system.");
        }
        else {
            @helpstring =
              ("The preceding error message comes from Perl.  Apparently,",
               "there’s a bug in your Perl code.  You’ll need to sort that",
               "out in your document and re-run perltex.");
        }
        my $helpstring = join ("\\MessageBreak\n", @helpstring);
        $helpstring =~ s/\.  /.\\space\\space /g;
        $result .= "{$helpstring}";
    }
```

Log the resulting LaTeX code.

```
    print LOGFILE "%" x 30, " LATEX RESULT ", "%" x 30, "\n";
    print LOGFILE $result, "\n\n";
```

We add `\endinput` to the generated LaTeX code to suppress an extraneous end-of-line character that TeX would otherwise insert.

```
    $result .= '\endinput';
```

Continuing the protocol described in Figure 4 on page 20 we now write `$result` (which contains either the result of executing the user's or a `\PackageError`) to the `$fromperl` file, delete `$toflag`, `$toperl`, and `$doneflag`, and notify LaTeX by touching the `$fromflag` file.

```
    open (FROMPERL, ">$fromperl") || die "open($fromperl): $!\n";
    syswrite FROMPERL, $result;
    close FROMPERL;

    unlink $toflag while -e $toflag;
    unlink $toperl while -e $toperl;
    unlink $doneflag while -e $doneflag;

    open (FROMFLAG, ">$fromflag") || die "open($fromflag): $!\n";
    close FROMFLAG;
```

We have to perform one final LaTeX-to-Perl synchronization step. Otherwise, a subsequent `\perl`[re]`newcommand` would see that `$fromflag` already exists and race ahead, finding that `$fromperl` does not contain what it's supposed to.

```
    $awaitexists->($toperl);
    unlink $fromflag while -e $fromflag;
    open (DONEFLAG, ">$doneflag") || die "open($doneflag): $!\n";
    close DONEFLAG;
}
```

### 3.2.9 Final cleanup

If we exit abnormally we should do our best to kill the child `latex` process so that it doesn't continue running forever, holding onto system resources.

```
END {
    close LOGFILE;
    if (defined $latexpid) {
        kill (9, $latexpid);
        exit 1;
    }
    exit 0;
}

__END__
```

### 3.2.10 `perltex.pl` POD documentation

`perltex.pl` includes documentation in Perl's POD (Plain Old Documentation) format. This is used both to produce manual pages and to provide usage information when `perltex.pl` is invoked with the `--help` option. The POD documentation is not listed here as part of the documented `perltex.pl` source code because it contains essentially the same information as that shown in Section 2.2. If you're curious what the POD source looks like then see the generated `perltex.pl` file.

## 3.3 Porting to other languages

Perl is a natural choice for a LaTeX macro language because of its excellent support for text manipulation including extended regular expressions, string interpolation, and "here" strings, to name a few nice features. However, Perl's syntax is unusual and its semantics are rife with annoying special cases. Some users will therefore long for a ⟨*some-language-other-than-Perl*⟩TeX. Fortunately, porting PerlTeX to use a different language should be fairly straightforward. `perltex.pl` will need to be rewritten in the target language, of course, but `perltex.sty` modifications will likely be fairly minimal. In all probability, only the following changes will need to be made:

- Rename `perltex.sty` and `perltex.pl` (and choose a package name other than "PerlTeX") as per the PerlTeX license agreement (Section 4).
- In your replacement for `perltex.sty`, replace all occurrences of "`plmac`" with a different string.
- In your replacement for `perltex.pl`, choose different file extensions for the various helper files.

The importance of these changes is that they help ensure version consistency and that they make it possible to run ⟨*some-language-other-than-Perl*⟩TeX alongside PerlTeX, enabling multiple programming languages to be utilized in the same LaTeX document.

## 4 License agreement

## Change History

v1.0
- General: Initial version . . . . . . . . 1

v1.0a
- General: Made all `unlink` calls wait for the file to actually disappear 27
  - Undefined `$/` only locally . . . . 28
- `$awaitexists`: Bug fix: Added "`undef $latexpid`" to make the `END` block correctly return a status code of 0 on success . . 28

v1.1
- General: Added new `\perlnewenvironment` and `\perlrenewenvironment` macros . . . . . . . . . . . . . . . . 17
- `\plmac@havecode`: Added a `\plmac@next` hook to support PerlTeX's new environment-defining macros . . . . . . . . . . . 15
- `\plmac@write@perl@i`: Added a dummy version of the macro to use if `latex` was launched directly, without `perltex.pl` . . 22
  - Made argument-handling more rational by making `\protect`, `\begin`, and `\end` non-expandable . . . . . . . . . . . . . . 21

v1.2
- General: Renamed `perlmacros.sty` to `perltex.sty` for consistency. 1
- `\plmac@write@perl@i`: Moved the `\input` of the generated Perl code to the end of the routine in order to support recursive PerlTeX macro invocations. . . 22

v1.3
- General: Modified `perltex.pl` to eschew the sandbox altogether when `--nosafe` is specified . . 24
- `\perldo`: Introduced `\perldo` to support code execution outside of all subroutines. . . . . . . . . . 19
- `\plmac@run@code`: Added to assist `\perldo` . . . . . . . . . . . . . . . . 19

## Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.