

# PerlTeX: Defining L<sup>A</sup>T<sub>E</sub>X macros in terms of Perl code\*

Scott Pakin  
scott+pt@pakin.org

August 12, 2007

## Abstract

PerlTeX is a combination Perl script (`perltex.pl`) and L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> style file (`perltex.sty`) that, together, give the user the ability to define L<sup>A</sup>T<sub>E</sub>X macros in terms of Perl code. Once defined, a Perl macro becomes indistinguishable from any other L<sup>A</sup>T<sub>E</sub>X macro. PerlTeX thereby combines L<sup>A</sup>T<sub>E</sub>X's typesetting power with Perl's programmability.

## 1 Introduction

T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X, the most popular macro package for T<sub>E</sub>X, does little to simplify T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X-based but contain a modicum of Perl. PerlTeX seamlessly integrates Perl code into a L<sup>A</sup>T<sub>E</sub>X document, enabling the user to define macros whose bodies consist of Perl code instead of T<sub>E</sub>X and L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X authors are sufficiently versed in the T<sub>E</sub>X 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  $\text{\LaTeX}$  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  $\text{\PerlTeX}$  makes it easy to export this to  $\text{\LaTeX}$ :

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

`\substr` can then be used just like any other  $\text{\LaTeX}$  macro—and as simply as Perl’s `substr` function:

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



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  $\text{\LaTeX}$  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}
```



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

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}{\thesrow\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}

```



	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 Perl<sub>T</sub>EX. First, documents must include a “`\usepackage{perltex}`” line in their preamble in order to define `\perlnewcommand`, `\perlrenewcommand`, `\perlnewenvironment`, and `\perlrenewenvironment`. Second, L<sup>A</sup>T<sub>E</sub>X documents must be compiled using the `perltex.pl` wrapper script.

### 2.1 Defining and redefining Perl macros

`\perlnewcommand` `perltex.sty` defines five macros: `\perlnewcommand`, `\perlrenewcommand`, `\perlnewenvironment`, `\perlrenewenvironment`, and `\perldo`. The first four of these behave exactly like their L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> counterparts—`\newcommand`, `\renewcommand`, `\newenvironment`, and `\renewenvironment`—except that the macro body consists of Perl code that dynamically generates L<sup>A</sup>T<sub>E</sub>X 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 Perl<sub>T</sub>EX-defined macro or environments is converted to a Perl subroutine named after the macro/environment but beginning with “`latex_`”. For example, a Perl<sub>T</sub>EX-defined L<sup>A</sup>T<sub>E</sub>X macro called `\myMacro` internally produces a Perl

subroutine called `latex_myMacro`. Macro arguments are converted to subroutine arguments. A  $\text{\LaTeX}$  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,  $\text{\PerlTeX}$  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 runtime 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  $\text{\PerlTeX}$  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
```



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

Macro arguments are expanded by  $\text{\LaTeX}$  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 `"\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 L<sup>A</sup>T<sub>E</sub>X, 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 L<sup>A</sup>T<sub>E</sub>X macros to be defined in terms of Perl code

## SYNOPSIS

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

## DESCRIPTION

L<sup>A</sup>T<sub>E</sub>X — through the underlying T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X’s typesetting strengths. **perltex** is the tool that enables a symbiosis between the two systems. All a user needs to do is compile a L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X’s `\newcommand` and `\renewcommand` except that the macro body contains Perl code instead of L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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 mechanisms 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 L<sup>A</sup>T<sub>E</sub>X compiler. The L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X to Perl

***jobname.frpl***

information sent from Perl to L<sup>A</sup>T<sub>E</sub>X

***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* Perl<sub>T</sub><sub>E</sub>X can skip Section 3, which presents the complete Perl<sub>T</sub><sub>E</sub>X source code. This section should be of interest primarily to those who wish to extend Perl<sub>T</sub><sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X side of Perl<sub>T</sub><sub>E</sub>X, and Section 3.2 presents the source code for `perltex.pl`, the Perl side of Perl<sub>T</sub><sub>E</sub>X. In toto, Perl<sub>T</sub><sub>E</sub>X consists of a relatively small amount of code. `perltex.sty` is only 224 lines of L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X trickery and is probably impenetrable to anyone who hasn't already tried his hand at L<sup>A</sup>T<sub>E</sub>X programming. Fortunately for the reader, the code is profusely commented so the aspiring L<sup>A</sup>T<sub>E</sub>X guru may yet learn something from it.

After documenting the `perltex.sty` and `perltex.pl` source code, a few suggestions are provided for porting Perl<sub>T</sub><sub>E</sub>X to use a backend language other than Perl (Section 3.3).

### 3.1 `perltex.sty`

Although I've written a number of L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X—code for later use
2. iterating over a macro's arguments

Storing non-L<sup>A</sup>T<sub>E</sub>X 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 `\catcoded` 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 T<sub>E</sub>X'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 `\relaxed` 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 L<sup>A</sup>T<sub>E</sub>X. `\plmac@tag` is a string of characters that should never occur within one of the user's macro names, macro arguments, or macro bodies. `perltx.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 L<sup>A</sup>T<sub>E</sub>X to Perl. `\plmac@fromfile` is the name of a file used for communication from Perl to L<sup>A</sup>T<sub>E</sub>X. `\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 `perltx.pl`.

Table 1: Variables used for communication between Perl and L<sup>A</sup>T<sub>E</sub>X

Variable	Purpose	<code>perltx.pl</code> assignment
<code>\plmac@tag</code>	<code>\plmac@tofile</code> field separator	(20 random letters)
<code>\plmac@tofile</code>	L <sup>A</sup> T <sub>E</sub> X → Perl communication	<code>\jobname.topl</code>
<code>\plmac@fromfile</code>	Perl → L <sup>A</sup> T <sub>E</sub> X communication	<code>\jobname.frpl</code>
<code>\plmac@toflag</code>	<code>\plmac@tofile</code> synchronization	<code>\jobname.tfpl</code>
<code>\plmac@fromflag</code>	<code>\plmac@fromfile</code> synchronization	<code>\jobname.ffpl</code>
<code>\plmac@doneflag</code>	<code>\plmac@fromflag</code> synchronization	<code>\jobname.dfpl</code>

```
\ifplmac@have@perltx
\plmac@have@perltxtrue
\plmac@have@perltxfalse
```

The following block of code checks the existence of each of the variables listed in Table 1. If any variable is not defined, `perltx.sty` gives an error message and—as we shall see on page 22—defines dummy versions of `\perl[re]newcommand` and `\perl[re]newenvironment`.

```
1 \newif\ifplmac@have@perltx
2 \plmac@have@perltxtrue
3 \@ifundefined{plmac@tag}{\plmac@have@perltxfalse}{}
4 \@ifundefined{plmac@tofile}{\plmac@have@perltxfalse}{}
5 \@ifundefined{plmac@fromfile}{\plmac@have@perltxfalse}{}
6 \@ifundefined{plmac@toflag}{\plmac@have@perltxfalse}{}
7 \@ifundefined{plmac@fromflag}{\plmac@have@perltxfalse}{}
8 \@ifundefined{plmac@doneflag}{\plmac@have@perltxfalse}{}
9 \ifplmac@have@perltx
10 \else
11   \PackageError{perltx}{Document must be compiled using perltx}
12     {Instead of compiling your document directly with latex, you need
13       to\MessageBreak use the perltx script. \space perltx sets up
14       a variety of macros needed by\MessageBreak the perltx
15       package as well as a listener process needed for\MessageBreak
16       communication between LaTeX and Perl.}
17 \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, `\perlido`.) 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 L<sup>A</sup>T<sub>E</sub>X 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 `\writeable` 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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
:
# <i>last</i>

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; perl<sub>tex</sub>.pl evaluates the code and writes \plmac@fromfile; finally, \plmac@write@perl \inputs \plmac@fromfile.

An example might help distinguish the myriad macros used internally by perl<sub>tex</sub>.sty. Consider the following call made by the user's document:

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

Table 2 shows how perl<sub>tex</sub>.sty parses that command into its constituent components and which components are bound to which perl<sub>tex</sub>.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` `\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  $\text{\LaTeX}$  code. Likewise, `\perlrenewcommand` is analogous to `\renewcommand` except that the macro body consists of Perl code instead of  $\text{\LaTeX}$  code. `\perlnewcommand` and `\perlrenewcommand` merely define `\plmac@command` and `\plmac@next` and invoke `\plmac@newcommand@i`.

```

18 \def\perlnewcommand{%
19   \let\plmac@command=\newcommand
20   \let\plmac@next=\relax
21   \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
22 }

23 \def\perlrenewcommand{%
24   \let\plmac@next=\relax
25   \let\plmac@command=\renewcommand
26   \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
27 }

```

`\plmac@newcommand@i` 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 `\writeable` (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`.

```

28 \def\plmac@newcommand@i#1#2{%
29   \ifx#1*%
30     \def\plmac@starchar{*}%
31   \else
32     \def\plmac@starchar{!}%
33   \fi
34   \def\plmac@macname{#2}%
35   \let\plmac@oldbody=#2\relax
36   \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
37     \expandafter\string\plmac@macname}%
38   \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%
39 }

```

`\plmac@newcommand@ii` `\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.

```

40 \def\plmac@newcommand@ii[#1]{%
41   \def\plmac@numargs{#1}%

```

```

42 \ifnextchar[{\plmac@newcommand@iii@opt}
43           {\plmac@newcommand@iii@no@opt}%]
44 }

```

```

\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`.

```

45 \def\plmac@newcommand@iii@opt[#1]{%
46   \def\plmac@defarg{#1}%
47   \plmac@haveargs
48 }

49 \def\plmac@newcommand@iii@no@opt{%
50   \let\plmac@defarg=\relax
51   \plmac@haveargs
52 }

```

```

\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 L<sup>A</sup>T<sub>E</sub>X 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.

```

53 \newtoks\plmac@perlcode

54 \def\plmac@haveargs{%
55   \begingroup
56   \let\do\@makeother\dospecials
57   \catcode'\^M=\active
58   \newlinechar'\^M
59   \endlinechar='\^M
60   \catcode'\{=1
61   \catcode'\}=2
62   \afterassignment\plmac@havecode
63   \global\plmac@perlcode

```

64 }

Control is transferred 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.

<code>\plmac@sep</code>	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 <code>\plmac@sep</code> as a carriage-return character of category code 11 ("letter"). 65 <code>{\catcode'\^M=11\gdef\plmac@sep{^M}}</code>
<code>\plmac@argnum</code>	Define a loop variable that will iterate from 1 to the number of arguments in the user's function, i.e., <code>\plmac@numargs</code> . 66 <code>\newcount\plmac@argnum</code>
<code>\plmac@havecode</code>	Now comes the final piece of what started as a call to <code>\perl[re]newcommand</code> . First, to reset all category codes back to normal, <code>\plmac@havecode</code> ends the group that was begun in <code>\plmac@haveargs</code> . 67 <code>\def\plmac@havecode{%</code> 68 <code>\endgroup</code>
<code>\plmac@define@sub</code>	We invoke <code>\plmac@write@perl</code> to define a Perl subroutine named after <code>\plmac@cleaned@macname</code> . <code>\plmac@define@sub</code> sends Perl the information shown in Figure 1 on page 13. 69 <code>\edef\plmac@define@sub{%</code> 70 <code>\noexpand\plmac@write@perl{DEF\plmac@sep</code> 71 <code>\plmac@tag\plmac@sep</code> 72 <code>\plmac@cleaned@macname\plmac@sep</code> 73 <code>\plmac@tag\plmac@sep</code> 74 <code>\the\plmac@perlcode</code> 75 <code>}%</code> 76 <code>}%</code> 77 <code>\plmac@define@sub</code>
<code>\plmac@body</code>	The rest of <code>\plmac@havecode</code> is preparation for defining the user's macro. ( $\text{\LaTeX 2}_{\epsilon}$ 's <code>\newcommand</code> or <code>\renewcommand</code> will do the actual work, though.) <code>\plmac@body</code> will eventually contain the complete ( $\text{\LaTeX}$ ) body of the user's macro. Here, we initialize it to the first three items listed in Figure 2 on page 13 (with intervening <code>\plmac@seps</code> ). 78 <code>\edef\plmac@body{%</code> 79 <code>USE\plmac@sep</code> 80 <code>\plmac@tag\plmac@sep</code> 81 <code>\plmac@cleaned@macname</code> 82 <code>}%</code>



`\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.

```

83 \let\plmac@hash=\relax
84 \plmac@argnum=\@ne
85 \loop
86   \ifnum\plmac@numargs<\plmac@argnum
87   \else
88     \edef\plmac@body{%
89       \plmac@body\plmac@sep\plmac@tag\plmac@sep
90       \plmac@hash\plmac@hash\number\plmac@argnum}%
91     \advance\plmac@argnum by \@ne
92   \repeat
93 \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`).

```

94 \expandafter\let\plmac@macname=\relax
95 \ifx\plmac@defarg\relax
96   \edef\plmac@define@command{%
97     \noexpand\plmac@command\plmac@starchar{\plmac@macname}%
98     [\plmac@numargs]{%
99       \noexpand\plmac@write@perl{\plmac@body}%
100    }%
101  }%
102 \else
103   \edef\plmac@define@command{%
104     \noexpand\plmac@command\plmac@starchar{\plmac@macname}%
105     [\plmac@numargs][\plmac@defarg]{%
106       \noexpand\plmac@write@perl{\plmac@body}%
107    }%
108  }%
109 \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`.

```
110 \expandafter\let\plmac@macname=\plmac@oldbody
111 \plmac@define@command
112 \plmac@next
113 }
```

### 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.

<pre>\perlnewenvironment \perlrenewenvironment   \plmac@command     \plmac@next</pre>	<p><code>\perlnewenvironment</code> and <code>\perlrenewenvironment</code> are the remaining two commands exported to the user by <code>perltex.sty</code>. <code>\perlnewenvironment</code> is analogous to <code>\newenvironment</code> except that the macro body consists of Perl code instead of <math>\text{\LaTeX}</math> code. Likewise, <code>\perlrenewenvironment</code> is analogous to <code>\renewenvironment</code> except that the macro body consists of Perl code instead of <math>\text{\LaTeX}</math> code. <code>\perlnewenvironment</code> and <code>\perlrenewenvironment</code> merely define <code>\plmac@command</code> and <code>\plmac@next</code> and invoke <code>\plmac@newenvironment@i</code>.</p>
---	---

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  $\text{\LaTeX}$  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.

```
114 \def\perlnewenvironment{%
115   \let\plmac@command=\newcommand
116   \let\plmac@next=\plmac@end@environment
117   \ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
118 }

119 \def\perlrenewenvironment{%
120   \let\plmac@command=\renewcommand
121   \let\plmac@next=\plmac@end@environment
122   \ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
123 }
```

<pre>\plmac@newenvironment@i   \plmac@starchar     \plmac@envname       \plmac@macname         \plmac@oldbody           \plmac@cleaned@macname</pre>	<p>The <code>\plmac@newenvironment@i</code> macro is analogous to <code>\plmac@newcommand@i</code>; see the description of <code>\plmac@newcommand@i</code> on page 14 to understand the ba-</p>
--	--

sic 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`.

```

124 \def\plmac@newenvironment@i#1#2{%
125   \ifx#1*%
126     \def\plmac@starchar{*}%
127   \else
128     \def\plmac@starchar{}%
129   \fi
130   \def\plmac@envname{#2}%
131   \expandafter\def\expandafter\plmac@macname\expandafter{\csname#2\endcsname}%
132   \expandafter\let\expandafter\plmac@oldbody\plmac@macname\relax
133   \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
134     \expandafter\string\plmac@macname}%
135   \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
136 }

```

`\plmac@end@environment` Recall that an environment definition is a shortcut for two macro definitions:  
`\plmac@next` `\langle name \rangle` and `\end<name>` (where `\langle name \rangle` was stored in `\plmac@envname` by  
`\plmac@macname` `\plmac@newenvironment@i`). After defining `\langle name \rangle`, `\plmac@havecode` trans-  
`\plmac@oldbody`fers control to `\plmac@end@environment` because `\plmac@next` was let-bound to  
`\plmac@cleaned@macname` `\plmac@end@environment` in `\perl[re]newenvironment`.

`\plmac@end@environment`'s purpose is to define `\end<name>`. This is a little tricky, however, because L<sup>A</sup>T<sub>E</sub>X'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`.

```

137 \def\plmac@end@environment{%
138   \expandafter\def\expandafter\plmac@next\expandafter{\expandafter
139     \let\csname end\plmac@envname\endcsname=\plmac@end@macro
140     \let\plmac@next=\relax
141   }%
142   \def\plmac@macname{\plmac@end@macro}%
143   \expandafter\let\expandafter\plmac@oldbody\csname end\plmac@envname\endcsname
144   \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
145     \expandafter\string\plmac@macname}%
146   \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
147 }

```

### 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 Perl<sub>T</sub>E<sub>X</sub> 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
<i>Ignored</i>
\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 L<sup>A</sup>T<sub>E</sub>X 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`.

```

148 \def\perldo{%
149   \begingroup
150   \let\do\@makeother\dospecials
151   \catcode'\^^M=\active
152   \newlinechar'\^^M
153   \endlinechar='\^^M
154   \catcode'\{=1
155   \catcode'\}=2
156   \afterassignment\plmac@have@run@code
157   \global\plmac@perlcode
158 }
```

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

```

159 \def\plmac@have@run@code{%
160   \endgroup
161   \edef\plmac@run@code{%
162     \noexpand\plmac@write@perl{RUN\plmac@sep
163       \plmac@tag\plmac@sep
164       N/A\plmac@sep
165       \plmac@tag\plmac@sep
166       \the\plmac@perlcode
167     }%
168   }%
169   \plmac@run@code
170 }
```

### 3.1.5 Communication between L<sup>A</sup>T<sub>E</sub>X 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, L<sup>A</sup>T<sub>E</sub>X-to-Perl communication is performed via the filesystem. In essence, L<sup>A</sup>T<sub>E</sub>X 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, L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X has finished writing Perl code and L<sup>A</sup>T<sub>E</sub>X needs to know when Perl has finished writing L<sup>A</sup>T<sub>E</sub>X code. The solution involves introducing three extra files—`\plmac@toflag`, `\plmac@fromflag`, and `\plmac@doneflag`—which are used exclusively for L<sup>A</sup>T<sub>E</sub>X-to-Perl synchronization.

There’s a catch: Although Perl can create and delete files, L<sup>A</sup>T<sub>E</sub>X can only create them. Even worse, L<sup>A</sup>T<sub>E</sub>X (more specifically, t<sub>E</sub>L<sub>A</sub>T<sub>E</sub>X, which is the T<sub>E</sub>X distribution under which I developed PerlT<sub>E</sub>X) cannot reliably poll for a file’s *nonexistence*; 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), PerlT<sub>E</sub>X should behave as expected.

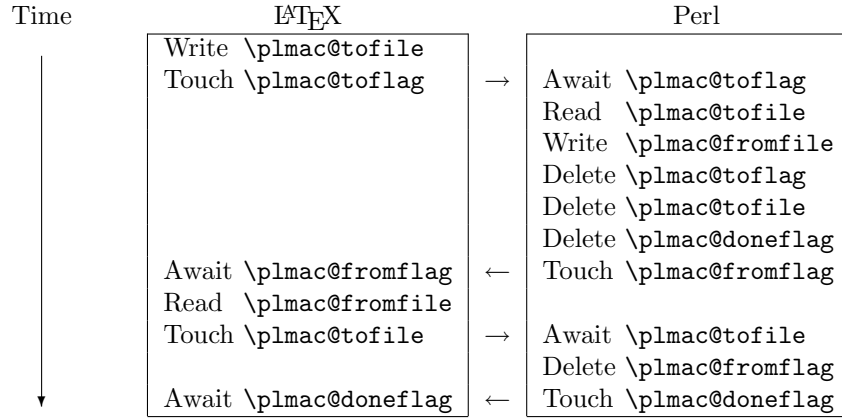


Figure 4: L<sup>A</sup>T<sub>E</sub>X-to-Perl communication protocol

```

\plmac@await@existence The purpose of the \plmac@await@existence macro is to repeatedly check
\ifplmac@file@exists the existence of a given file until the file actually exists. For convenience,
\plmac@file@existstrue we use LATEX 2ε’s \IfFileExists macro to check the file and invoke
\plmac@file@existsfalse \plmac@file@existstrue or \plmac@file@existsfalse, as appropriate.

171 \newif\ifplmac@file@exists
172 \newcommand{\plmac@await@existence}[1]{%
173   \loop
174     \IfFileExists{#1}%

```

```

175             {\plmac@file@existstrue}%
176             {\plmac@file@existsfalse}%
177     \ifplmac@file@exists
178     \else
179     \repeat
180 }

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

\plmac@write@perl \plmac@write@perl begins the LATEX-to-Perl data exchange, following the pro-
tocol 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.
182 \newcommand{\plmac@write@perl}{%
183     \begingroup
184     \let\do\@makeother\dospecials
185     \catcode'\^M=\active
186     \newlinechar'\^M
187     \endlinechar='\^M
188     \catcode'\{=1
189     \catcode'\}=2
190     \plmac@write@perl@i
191 }

\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 con-
sisting 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.
192 \ifplmac@have@perltex
193 \newcommand{\plmac@write@perl@i}[1]{%
    The first step is to write argument #1 to \plmac@tofile:
194     \immediate\openout\plmac@outfile=\plmac@tofile\relax
195     \let\protect=\noexpand
196     \def\begin{\noexpand\begin}%
197     \def\end{\noexpand\end}%
198     \immediate\write\plmac@outfile{#1}%
199     \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.

```
200 \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`.

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

```
202 \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`.

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

At this point, `\plmac@fromfile` should contain valid L<sup>A</sup>T<sub>E</sub>X code. However, we defer inputting it until we the very end. Doing so enables recursive and mutually recursive invocations of PerlT<sub>E</sub>X macros.

Because T<sub>E</sub>X can't delete files we require an additional L<sup>A</sup>T<sub>E</sub>X-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`.

```
204 \immediate\openout\plmac@outfile=\plmac@tofile\relax
```

```
205 \immediate\closeout\plmac@outfile
```

```
206 \plmac@await@existence\plmac@doneflag
```

The only thing left to do is to `\input` and evaluate `\plmac@fromfile`, which contains the L<sup>A</sup>T<sub>E</sub>X output from the Perl subroutine.

```
207 \input\plmac@fromfile\relax
```

```
208 }
```

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 “`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 doing nothing macro and redefine it as “`\fbox{Perl\TeX}`” at the `\begin{document}`.

```
209 \else
```

```
210 \newcommand{\plmac@write@perl@i}[1]{\endgroup}
```

```
211 \AtBeginDocument{%
```

```
212 \renewcommand{\plmac@write@perl@i}[1]{%
```

`\plmac@show@placeholder` There's really no point in outputting a framed “PerlT<sub>E</sub>X” 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 “PerlT<sub>E</sub>X” will be output.

```

213     \def\plmac@show@placeholder##1##2\@empty{%
214         \ifx##1D\relax
215             \endgroup
216         \else
217             \endgroup
218             \fbox{Perl\TeX}%
219         \fi
220     }%
221     \plmac@show@placeholder#1\@empty
222 }%
223 }
224 \fi

```

## 3.2 perltex.pl

`perltex.pl` is a wrapper script for `latex` (or any other  $\text{\LaTeX}$  compiler). It sets up client-server communication between  $\text{\LaTeX}$  and Perl, with  $\text{\LaTeX}$  as the client and Perl as the server. When a  $\text{\LaTeX}$  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  $\text{\LaTeX}$  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.

```

225 #! /usr/bin/env perl
226
227 #####
228 # Prepare a LaTeX run for two-way communication with Perl #
229 # By Scott Pakin <scott+pt@pakin.org>                      #
230 #####
231
232 #-----
233 # This is file 'perltex.pl',
234 # generated with the docstrip utility.
235 #
236 # The original source files were:
237 #
238 # perltex.dtx (with options: 'perltex')
239 #
240 # This is a generated file.
241 #
242 # Copyright (C) 2007, Scott Pakin <scott+pt@pakin.org>
243 #
244 # This file may be distributed and/or modified under the conditions
245 # of the LaTeX Project Public License, either version 1.3c of this
246 # license or (at your option) any later version. The latest
247 # version of this license is in:

```



```

248 #
249 #   http://www.latex-project.org/lppl.txt
250 #
251 # and version 1.3c or later is part of all distributions of LaTeX
252 # version 2006/05/20 or later.
253 #-----
254

```

### 3.2.2 Top-level code evaluation

In previous versions of `perltx.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.

```

255 sub top_level_eval ($)
256 {
257     return eval $_[0];
258 }

```

### 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.

```

259 use Safe;
260 use Opcode;
261 use Getopt::Long;
262 use Pod::Usage;
263 use File::Basename;
264 use POSIX;
265 use warnings;
266 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

<code>\$latexprog</code>	<code>\$latexprog</code> is the name of the L <sup>A</sup> T <sub>E</sub> X executable (e.g., “ <code>latex</code> ”).
<code>\$runsafely</code>	If <code>\$runsafely</code> 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.
<code>@permittedops</code>	<code>@permittedops</code> 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`.

```
267 my $latexprog;
268 my $runsafely = 1;
269 my @permittedops;
```

### Other global variables

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

```
270 my $progname = basename $0;
271 my $jobname = "texput";
272 my @latexcmdline;
273 my $topperl;
274 my $fromperl;
275 my $toflag;
276 my $fromflag;
277 my $doneflag;
278 my $logfile;
279 my $sandbox = new Safe;
280 my $sandbox_eval;
281 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.

```
282 $latexprog = $ENV{"PERLTEX"} || "latex";
283 Getopt::Long::Configure("require_order", "pass_through");
284 GetOptions("help"      => sub {pod2usage(-verbose => 1)}),
```

```

285         "latex=s" => \$latexprog,
286         "safe!"    => \$runsafely,
287         "permit=s" => \@permittedops) || pod2usage(2);

```

### Preparing a L<sup>A</sup>T<sub>E</sub>X command line

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

```

288 @latexcmdline = @ARGV;
289 my $firstcmd = 0;
290 for ($firstcmd=0; $firstcmd<=$#latexcmdline; $firstcmd++) {
291     my $option = $latexcmdline[$firstcmd];
292     next if substr($option, 0, 1) eq "-";
293     if (substr ($option, 0, 1) ne "\\") {
294         $jobname = basename $option, ".tex" ;
295         $latexcmdline[$firstcmd] = "\\input $option";
296     }
297     last;
298 }
299 push @latexcmdline, "" if $#latexcmdline==-1;

```

**\$separator** To avoid conflicts with the code and parameters passed to Perl from L<sup>A</sup>T<sub>E</sub>X (see Figure 1 on page 13 and Figure 2 on page 13) we define a separator string, **\$separator**, containing 20 random uppercase letters.

```

300 my $separator = "";
301 foreach (1 .. 20) {
302     $separator .= chr(ord("A") + rand(26));
303 }

```

Now that we have the name of the L<sup>A</sup>T<sub>E</sub>X job (**\$jobname**) we can assign **\$toperl**, **\$fromperl**, **\$toflag**, **\$fromflag**, **\$doneflag**, and **\$logfile** in terms of **\$jobname** plus a suitable extension.

```

304 $toperl = $jobname . ".topl";
305 $fromperl = $jobname . ".frpl";
306 $toflag = $jobname . ".tfpl";
307 $fromflag = $jobname . ".ffpl";
308 $doneflag = $jobname . ".dfpl";
309 $logfile = $jobname . ".lgpl";

```

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

```

310 $latexcmdline[$firstcmd] =
311     sprintf '\makeatletter' . '\def%s{%s}' x 6 . '\makeatother%s',
312     '\plmac@tag', $separator,
313     '\plmac@tofile', $toperl,
314     '\plmac@fromfile', $fromperl,
315     '\plmac@toflag', $toflag,
316     '\plmac@fromflag', $fromflag,

```

```

317     '\plmac@doneflag', $doneflag,
318     $latexcmdline[$firstcmd];

```

### 3.2.6 Launching L<sup>A</sup>T<sub>E</sub>X

We start by deleting the `$topperl`, `$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.

```

319 foreach my $file ($topperl, $fromperl, $toflag, $fromflag, $doneflag) {
320     unlink $file while -e $file;
321 }
322 open (LOGFILE, ">$logfile") || die "open(\"$logfile\"): $!\n";
323 defined ($latexpid = fork) || die "fork: $!\n";
324 unshift @latexcmdline, $latexprog;
325 if (!$latexpid) {
326     exec {@latexcmdline[0]} @latexcmdline;
327     die "exec('@latexcmdline'): $!\n";
328 }

```

### 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 L<sup>A</sup>T<sub>E</sub>X. 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.

```

329 if ($runsafely) {
330     @permittedops=(":browse") if $#permittedops==--1;
331     $sandbox->permit_only (@permittedops);
332     $sandbox_eval = sub {@sandbox->reval($_[0])};
333 }
334 else {
335     $sandbox_eval = \&top_level_eval;
336 }

```

### 3.2.8 Communicating with L<sup>A</sup>T<sub>E</sub>X

The following code constitutes `perltex.pl`'s main loop. Until `latex` exits, the loop repeatedly reads Perl code from L<sup>A</sup>T<sub>E</sub>X, evaluates it, and returns the result

as per the protocol described in Figure 4 on page 21.

```
337 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.

```
338     my $awaitexists = sub {
339         while (!-e $_[0]) {
340             sleep 0;
341             if (waitpid($latexpid, &WNOHANG)==-1) {
342                 foreach my $file ($toperl, $fromperl, $toflag,
343                     $fromflag, $doneflag) {
344                     unlink $file while -e $file;
345                 }
346                 undef $latexpid;
347                 exit 0;
348             }
349         }
350     };
```

**\$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**.

```
351     $awaitexists->($toflag);
352     my $entirefile;
353     {
354         local $/ = undef;
355         open (TOPERL, "<$toperl") || die "open($toperl): !\n";
356         $entirefile = <TOPERL>;
357         close TOPERL;
358     }
```

**\$optag** We split the contents of **\$entirefile** into an operation tag (either DEF, USE, **\$macroname** 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.

```
359     my ($optag, $macroname, @otherstuff) =
360         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.

```
361     $macroname =~ s/^[^A-Za-z]+//;
362     $macroname =~ s/\W/_/g;
363     $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 “\,”.

```

364     if ($optag eq "USE") {
365         foreach (@otherstuff) {
366             s/\\/\\\\/g;
367             s/\'/\\\'/g;
368             $_ = "\$_";
369         }
370     }

```

`$perlcode` There are three possible values that can be assigned to `$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 Perl<sub>T</sub><sub>E</sub><sub>X</sub> macro definition into a Perl subroutine definition and Figure 6 presents an example of how the following code converts a Perl<sub>T</sub><sub>E</sub><sub>X</sub> macro invocation into a Perl subroutine invocation.

```

371     my $perlcode;
372     if ($optag eq "DEF") {
373         $perlcode =
374             sprintf "sub %s {%s}\n",
375                 $macroname, $otherstuff[0];
376     }
377     elsif ($optag eq "USE") {
378         $perlcode = sprintf "%s (%s);\n", $macroname, join(" ", @otherstuff);
379     }
380     elsif ($optag eq "RUN") {
381         $perlcode = $otherstuff[0];
382     }

```

LaTeX: 

<pre>\perlnewcommand{\mymacro}[2]{%     sprintf "Isn't \$_[0] %s \$_[1]?\\n",     \$_[0]&gt;= \$_[1] ? "&gt;=" : "&lt;" }</pre>
---



Perl: 

<pre>sub latex_mymacro {     sprintf "Isn't \$_[0] %s \$_[1]?\\n",     \$_[0]&gt;= \$_[1] ? "&gt;=" : "&lt;" }</pre>
--

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

LaTeX: `\mymacro{12}{34}`



Perl: `latex_mymacro ('12', '34');`

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

```

383     else {
384         die "${progname}: Internal error -- unexpected operation tag \"${optag}\"\\n";
385     }

```

Log what we're about to evaluate.

```

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

```

**\$result** We're now ready to execute the user's code using the `$sandbox_eval` function.

**\$msg** If a warning occurs we write it as a Perl comment to the log file. If an error occurs (i.e., `$@` is defined) we replace the result (**\$result**) with a call to LaTeX 2<sub>ε</sub>'s `\PackageError` macro to return a suitable error message. We produce one error message for sandbox policy violations (detected by the error message, `$@`, 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>*”.

```

388     undef $_;
389     my $result;
390     {
391         my $warningmsg;
392         local $SIG{__WARN__} =
393             sub {chomp ($warningmsg=$_[0]); return 0};
394         $result = $sandbox_eval->($perlcode);
395         if (defined $warningmsg) {
396             $warningmsg =~ s/at \ (eval \d+\) line \d+\W+//;
397             print LOGFILE "# ==> $warningmsg\\n\\n";
398         }
399     }
400     $result="" if !$result || $optag eq "RUN";
401     if ($@) {
402         my $msg = $@;
403         $msg =~ s/at \ (eval \d+\) line \d+\W+//;
404         $msg =~ s/\\s+//;
405         $result = "\\PackageError{perltex}{$msg}";
406         my @helpstring;
407         if ($msg =~ /\btrapped by\b/) {
408             @helpstring =
409                 ("The preceding error message comes from Perl. Apparently,",
410                  "the Perl code you tried to execute attempted to perform an",

```

```

411         "‘unsafe’ operation. If you trust the Perl code (e.g., if",
412         "you wrote it) then you can invoke perltex with the --nosafe",
413         "option to allow arbitrary Perl code to execute.",
414         "Alternatively, you can selectively enable Perl features",
415         "using perltex’s --permit option. Don’t do this if you don’t",
416         "trust the Perl code, however; malicious Perl code can do a",
417         "world of harm to your computer system.");
418     }
419     else {
420         @helpstring =
421             ("The preceding error message comes from Perl. Apparently,",
422             "there’s a bug in your Perl code. You’ll need to sort that",
423             "out in your document and re-run perltex.");
424     }
425     my $helpstring = join ("\\MessageBreak\\n", @helpstring);
426     $helpstring =~ s/\. /\.\.\space\space /g;
427     $result .= "{$helpstring}";
428 }

```

Log the resulting L<sup>A</sup>T<sub>E</sub>X code.

```

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

```

We add `\endinput` to the generated L<sup>A</sup>T<sub>E</sub>X code to suppress an extraneous end-of-line character that T<sub>E</sub>X would otherwise insert.

```

431     $result .= '\endinput';

```

Continuing the protocol described in Figure 4 on page 21 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 `$doneflag`, and notify L<sup>A</sup>T<sub>E</sub>X by touching the `$fromflag` file.

```

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

435     unlink $toflag while -e $toflag;
436     unlink $toperl while -e $toperl;
437     unlink $doneflag while -e $doneflag;

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

```

We have to perform one final L<sup>A</sup>T<sub>E</sub>X-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.

```

440     $awaitexists->($toperl);
441     unlink $fromflag while -e $fromflag;
442     open (DONEFLAG, ">$doneflag") || die "open($doneflag): $!\n";
443     close DONEFLAG;
444 }

```



### 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.

```
445 END {  
446     close LOGFILE;  
447     if (defined $latexpid) {  
448         kill (9, $latexpid);  
449         exit 1;  
450     }  
451     exit 0;  
452 }  
453  
454 __END__
```

### 3.2.10 perl<sub>tex</sub>.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  $\text{\LaTeX}$  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)* $\text{\TeX}$ . Fortunately, porting Perl $\text{\TeX}$  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 “Perl $\text{\TeX}$ ”) as per the Perl $\text{\TeX}$  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)* $\text{\TeX}$  alongside Perl $\text{\TeX}$ , enabling multiple programming languages to be utilized in the same  $\text{\LaTeX}$  document.

## 4 License agreement

Copyright © 2007, Scott Pakin <scott+pt@pakin.org>

These files may be distributed and/or modified under the conditions of the L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X version 2006/05/20 or later.

## Change History

v1.0	General: Initial version . . . . .	1	Made argument-handling more rational by making <code>\protect</code> , <code>\begin</code> , and <code>\end</code> non-expandable . . . . .	22	
v1.0a	General: Made all <code>\unlink</code> calls wait for the file to actually disappear	28			
	Undefined <code>\$/</code> only locally . . . .	29			
	<code>\$awaitexists</code> : Bug fix: Added “ <code>\undef \$latexpid</code> ” to make the <code>END</code> block correctly return a status code of 0 on success . .	29			
v1.1	General: Added new <code>\perlnewenvironment</code> and <code>\perlrenewenvironment</code> macros . . . . .	18			
	<code>\plmac@havecode</code> : Added a <code>\plmac@next</code> hook to support PerlT <sub>E</sub> X’s new environment-defining macros . . . . .	16			
	<code>\plmac@write@perl@i</code> : Added a dummy version of the macro to use if <code>latex</code> was launched directly, without <code>perltex.pl</code> . .	23			
			v1.2	General: Renamed <code>perlmacros.sty</code> to <code>perltex.sty</code> for consistency. . .	1
				<code>\plmac@write@perl@i</code> : Moved the <code>\input</code> of the generated Perl code to the end of the routine in order to support recursive PerlT <sub>E</sub> X macro invocations. . .	23
			v1.3	General: Modified <code>perltex.pl</code> to eschew the sandbox altogether when <code>--nosafe</code> is specified . .	25
				<code>\perl@do</code> : Introduced <code>\perl@do</code> to support code execution outside of all subroutines. . . . .	20
				<code>\plmac@run@code</code> : Added to assist <code>\perl@do</code> . . . . .	20

## 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.

Symbols	A
<code>\\$</code> . . . . .	285, 286
<code>\&amp;</code> . . . . .	335
<code>\{</code> . . . . .	60, 154, 188
<code>\}</code> . . . . .	61, 155, 189
<code>\^</code> . . . . .	57–59, 65, 151–153, 185–187
<code>\afterassignment</code> . . . . .	62, 156
<code>\AtBeginDocument</code> . . . . .	211
<code>\$awaitexists</code> . . . . .	<u>338</u>

<b>C</b>		\perldo	148
\closeout	199, 202, 205	\perlnewcommand	18
<b>D</b>		\perlnewenvironment	114
\do	56, 150, 184	\perlrenewcommand	18
\$doneflag	270	\perlrenewenvironment	114
\dospecials	56, 150, 184	@permittedops	267
<b>E</b>		\plmac@argnum	66, 84, 86, 90, 91
\endinput	431	\plmac@await@existence	171, 203, 206
\endlinechar	59, 153, 187	\plmac@body	78
\$entirefile	351	\plmac@cleaned@macname	28, 72, 81, 124, 137
<b>F</b>		\plmac@command	18, 97, 104, 114
\fbox	218	\plmac@defarg	45, 95, 105
\$firstcmd	288	\plmac@define@command	94
\$fromflag	270	\plmac@define@sub	69
\$fromperl	270	\plmac@doneflag	206, 317
<b>I</b>		\plmac@end@environment	116, 121, 137
\IfFileExists	174	\plmac@end@macro	139, 142
\ifplmac@file@exists	171	\plmac@envname	124, 139, 143
\ifplmac@have@perltx	1, 192	\plmac@file@existsfalse	171
\input	207	\plmac@file@existstrue	171
<b>J</b>		\plmac@fromfile	207, 314
\$jobname	270	\plmac@fromflag	203, 316
<b>L</b>		\plmac@hash	83
@latexcmdline	270	\plmac@have@perltxfalse	1
\$latexpid	270	\plmac@have@perltxtrue	1
\$latexprog	267	\plmac@have@run@code	156, 159
\$logfile	270	\plmac@haveargs	47, 51, 53
<b>M</b>		\plmac@havecode	62, 67
\$macroname	359	\plmac@macname	28, 94, 97, 104, 110, 124, 137
\$msg	388	\plmac@newcommand@i	21, 26, 28
<b>N</b>		\plmac@newcommand@ii	38, 40, 135, 146
\newcommand	19, 115, 172, 182, 193, 210	\plmac@newcommand@iii@no@opt	43, 45
\newlinechar	58, 152, 186	\plmac@newcommand@iii@opt	42, 45
<b>O</b>		\plmac@newenvironment@i	117, 122, 124
\openout	194, 201, 204	\plmac@next	18, 112, 114, 137
\$optag	359	\plmac@numargs	40, 86, 98, 105
\$option	288	\plmac@oldbody	28, 110, 124, 137
@otherstuff	359	\plmac@outfile	181, 194, 198, 199, 201, 202, 204, 205
<b>P</b>		\plmac@perlcode	53, 74, 157, 166
\PackageError	11	\plmac@run@code	159
\$perlcode	371	\plmac@sep	65, 70–73, 79, 80, 89, 162–165
		\plmac@show@placeholder	213
		\plmac@starchar	28, 97, 104, 124
		\plmac@tag	71, 73, 80, 89, 163, 165, 312
		\plmac@tofile	194, 204, 313
		\plmac@toflag	201, 315
		\plmac@write@perl	70, 99, 106, 162, 182

\plmac@write@perl@i .....	190, <u>192</u>	\$sandbox_eval .....	<u>270</u>
\$progname .....	<u>270</u>	\$separator .....	<u>300</u>
<b>R</b>		<b>T</b>	
\renewcommand .....	25, 120, <u>212</u>	\$toflag .....	<u>270</u>
\$result .....	<u>388</u>	\$toperl .....	<u>270</u>
\$runsafely .....	<u>267</u>		
<b>S</b>		<b>W</b>	
\$sandbox .....	<u>270</u>	\write .....	198