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

Scott Pakin

scott+pt@pakin.org

July 10, 2010

**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 v2.1, dated 2010/07/10.

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}
```

$$\Downarrow$$

2

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

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}{\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 2.3.) 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 Making `perltex.pl` optional

Normally, `perltex.sty` issues a `Document must be compiled using perltex` error if a document specifies `\usepackage{perltex}` but is not compiled using `perltex.pl`. However, sometimes PerlTeX may be needed merely to enhance a

document's formatting without being mandatory for compiling the document. For

`optional` such cases, the `optional` package option instructs `perltex.sty` only to note that `Document was compiled without using the perltex script` without abort-

`\ifperl` ing the compilation. The author can then use the `\ifperl` macro to test if `perltex.pl` is being used and, if not, provide alternative definitions for macros and environments defined with `\perlnewcommand` and `\perlnewenvironment`.

See Section 2.4 for a large PerlTeX example that uses `optional` and `\ifperl` to define an environment one way if `perltex.pl` is detected and another way if not. The text preceding the example also shows how to enable a document to compile even if `perltex.sty` is not even installed.

## 2.3 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*] [–**makesty**] [*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 *Safe* 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 *Opcode* for more information.

**--makesty**

> Generate a LaTeX style file called *noperltex.sty*. Replacing the document's `\usepackage{perltex}` line with `\usepackage{noperltex}` produces the same output but does not require PerlTeX, making the document suitable for distribution to people who do not have PerlTeX installed. The disadvantage is that *noperltex.sty* is specific to the document that produced it. Any changes to the document's PerlTeX macro definitions or macro invocations necessitates rerunning **perltex** with the **--makesty** option.

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 OPTIONS) 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

***noperltex-#.tex***

      file generated by *noperltex.sty* for each PerlTeX macro invocation

## NOTES

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

## SEE ALSO

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

## AUTHOR

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

## 2.4 A large, complete example

Suppose we want to define a `linkwords` environment that exhibits the following characteristics:

1. All words that appear within the environment's body are automatically hyperlinked to a given URL that incorporates the lowercase version of the word somewhere within that URL.

2. The environment accepts an optional list of stop words that should not be hyperlinked.

3. Paragraph breaks, nested environments, and other LaTeX markup are allowed within the environment's body.

Because of the reliance on text manipulation (parsing the environment's body into words, comparing each word against the list of stop words, distinguishing between text and LaTeX markup, etc.), these requirements would be difficult to meet without PerlTeX.

We use three packages to help define the `linkwords` environment: `perltex` for text manipulation, `hyperref` for creating hyperlinks, and `environ` for gathering up the body of an environment and passing it as an argument to a macro. Most of the work is performed by the PerlTeX macro `\dolinkwords`, which takes three arguments: a URL template that contains "`\%s`" as a placeholder for a word from the text, a mandatory but possibly empty space-separated list of lowercase stop words, and the input text to process. `\dolinkwords` first replaces all sequences of the form \⟨*letters*⟩, `\begin{`⟨*letters*⟩`}`, or `\end{`⟨*letters*⟩`}` with dummy alphanumerics but remembers which dummy sequence corresponds with each original LaTeX sequence. The macro then iterates over each word in the input text, formatting each non-stop-word using the URL template. Contractions (words containing apostrophes) are ignored. Finally, `\dolinkwords` replaces the dummy sequences with the corresponding LaTeX text and returns the result.

The `linkwords` environment itself is defined using the `\NewEnviron` macro from the `environ` package. With `\NewEnviron`'s help, `linkwords` accumulates its body into a `\BODY` macro and passes that plus the URL template and the optional list of stop words to `\dolinkwords`.

As an added bonus, `\ifperl`...`\else`...`\fi` is used to surround the definition of the `\dolinkwords` macro and `linkwords` environment. If the document is not run through `perltex.pl`, `linkwords` is defined as a do-nothing environment that simply typesets its body as is. Note that `perltex.sty` is loaded with the `optional` option to indicate that the document can compile without `perltex.pl`. However, the user still needs `perltex.sty` to avoid getting a `File 'perltex.sty' not found` error from LaTeX. To produce a document that can compile even without `perltex.sty` installed, replace the `\usepackage[optional]{perltex}` line with

the following LaTeX code:

```
\IfFileExists{perltex.sty}
            {\usepackage[optional]{perltex}}
            {\newif\ifperl}
```

A complete LaTeX document is presented below. This document, which includes the definition and a use of the linkwords environment, can be extracted from the PerlTeX source code into a file called example.tex by running

```
tex perltex.ins
```

In the following listing, line numbers are suffixed with "X" to distinguish them from line numbers associated with PerlTeX's source code.

```
1X \documentclass{article}
2X \usepackage[optional]{perltex}
3X \usepackage{environ}
4X \usepackage{hyperref}
5X
6X \ifperl
7X
8X  \perlnewcommand{\dolinkwords}[3]{
9X      # Preprocess our arguments.
10X     $url = $_[0];
11X     $url =~ s/\\\%s/\%s/g;
12X     %stopwords = map {lc $_ => 1} split " ", $_[1];
13X     $stopwords{""} = 1;
14X     $text = $_[2];
15X
16X     # Replace LaTeX code in the text with placeholders.
17X     $placeholder = "ABCxyz123";
18X     %substs = ();
19X     $replace = sub {$substs{$placeholder} = $_[0]; $placeholder++};
20X     $text =~ s/\\(begin|end)\s+\{[a-z]+\}/$replace->($&)/gse;
21X     $text =~ s/\\[a-z]+/$replace->($&)/gse;
22X
23X     # Hyperlink each word that's not in the stop list.
24X     $newtext = "";
25X     foreach $word (split /((?<=[-\A\s])[\'a-z]+\b)/i, $text) {
26X         $lcword = lc $word;
27X         if (defined $stopwords{$lcword} || $lcword =~ /[^a-z]/) {
28X             $newtext .= $word;
29X         }
30X         else {
31X             $newtext .= sprintf "\\href{$url}{%s}", $lcword, $word;
32X         }
33X     }
```

```
34X
35X        # Restore original text from placeholders and return the new text.
36X        while (($tag, $orig) = each %substs) {
37X            $newtext =~ s/\Q$tag\E/$orig/gs;
38X        }
39X        return $newtext;
40X    }
41X
42X    \NewEnviron{linkwords}[2][]{\dolinkwords{#2}{#1}{\BODY}}{}
43X
44X \else
45X
46X    \newenvironment{linkwords}[2][]{}{}
47X
48X \fi
49X
50X \begin{document}
51X
52X \newcommand{\stopwords}{a an the of in am and or but i we me you us them}
53X
54X \begin{linkwords}[\stopwords]{http://www.google.com/search?q=define:\%s}
55X \begin{verse}
56X    I'm very good at integral and differential calculus; \\
57X    I know the scientific names of beings animalculous:  \\
58X    In short, in matters vegetable, animal, and mineral, \\
59X    I am the very model of a modern Major-General.
60X \end{verse}
61X \end{linkwords}
62X
63X \end{document}
```

## 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 301 lines of LaTeX and `perltex.pl` is only 329 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 sug-

gestions 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

`\ifplmac@required`
`\plmac@requiredtrue`
`\plmac@requiredfalse`

The `optional` package option lets an author specify that the document can be built successfully even without PerlTeX. Typically, this means that the document uses `\ifperl` to help define reduced-functionality equivalents of any document-defined PerlTeX macros and environments. When `optional` is not specified, `perltex.sty` issues an error message if the document is compiled without using `perltex.pl`. When `optional` is specified, `perltex.sty` suppresses the error message.

```
64 \newif\ifplmac@required \plmac@requiredtrue
65 \DeclareOption{optional}{\plmac@requiredfalse} \ProcessOptions\relax
```

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 |

\ifperl
\perltrue
\perlfalse

The following block of code checks the existence of each of the variables listed in Table 1 plus \plmac@pipe, a Unix named pipe used for to improve performance. If any variable is not defined, perltex.sty gives an error message and—as we shall see on page 26—defines dummy versions of \perl[re]newcommand and \perl[re]newenvironment.

```
66 \newif\ifperl
67 \perltrue
68 \@ifundefined{plmac@tag}{\perlfalse}{}
69 \@ifundefined{plmac@tofile}{\perlfalse}{}
70 \@ifundefined{plmac@fromfile}{\perlfalse}{}
71 \@ifundefined{plmac@toflag}{\perlfalse}{}
72 \@ifundefined{plmac@fromflag}{\perlfalse}{}
73 \@ifundefined{plmac@doneflag}{\perlfalse}{}
74 \@ifundefined{plmac@pipe}{\perlfalse}{}
75 \ifperl
76 \else
77   \ifplmac@required
78     \PackageError{perltex}{Document must be compiled using perltex}
79       {Instead of compiling your document directly with latex, you need
80        to\MessageBreak use the perltex script.  \space perltex sets up
81        a variety of macros needed by\MessageBreak the perltex
82        package as well as a listener process needed for\MessageBreak
83        communication between LaTeX and Perl.}
84     \else
85       \bgroup
86        \obeyspaces
87        \typeout{perltex: Document was compiled without using the perltex script;}
88        \typeout{        it may not print as desired.}
89       \egroup
90   \fi
91 \fi
```

14

### 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 |

$$\vdots$$

| #$\langle last \rangle$ |
| --- |

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 \inputs \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<br>\perlrenewcommand<br>\plmac@command<br>\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. |

```
92 \def\perlnewcommand{%
93   \let\plmac@command=\newcommand
94   \let\plmac@next=\relax
95   \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
96 }
```

```
97 \def\perlrenewcommand{%
98   \let\plmac@next=\relax
99   \let\plmac@command=\renewcommand
100   \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
101 }
```

| | |
|---|---|
| \plmac@newcommand@i<br>\plmac@starchar<br>\plmac@macname<br>\plmac@oldbody<br>\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 \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. |

```
102 \def\plmac@newcommand@i#1#2{%
103   \ifx#1*%
104     \def\plmac@starchar{*}%
105   \else
106     \def\plmac@starchar{}%
107   \fi
108   \def\plmac@macname{#2}%
109   \let\plmac@oldbody=#2\relax
110   \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
111     \expandafter\string\plmac@macname}%
112   \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
113 }
```

| | |
|---|---|
| \plmac@newcommand@ii<br>\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. |

```
114 \def\plmac@newcommand@ii[#1]{%
115   \def\plmac@numargs{#1}%
```

```
116    \@ifnextchar[{\plmac@newcommand@iii@opt}
117                {\plmac@newcommand@iii@no@opt}%]
118 }
```

\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.

```
119 \def\plmac@newcommand@iii@opt[#1]{%
120    \def\plmac@defarg{#1}%
121    \plmac@haveargs
122 }
```

```
123 \def\plmac@newcommand@iii@no@opt{%
124    \let\plmac@defarg=\relax
125    \plmac@haveargs
126 }
```

\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.

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

```
128 \def\plmac@haveargs{%
129    \begingroup
130       \let\do\@makeother\dospecials
131       \catcode'\^^M=\active
132       \newlinechar'\^^M
133       \endlinechar='\^^M
134       \catcode'\{=1
135       \catcode'\}=2
136       \afterassignment\plmac@havecode
137       \global\plmac@perlcode
```

```
138 }
```

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").

```
139 {\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`.

```
140 \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`.

```
141 \def\plmac@havecode{%
142   \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 16.

```
143   \edef\plmac@define@sub{%
144     \noexpand\plmac@write@perl{DEF\plmac@sep
145       \plmac@tag\plmac@sep
146       \plmac@cleaned@macname\plmac@sep
147       \plmac@tag\plmac@sep
148       \the\plmac@perlcode
149     }%
150   }%
151   \plmac@define@sub
```

`\plmac@body`    The rest of `\plmac@havecode` is preparation for defining the user's macro. (LATEX 2ε'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 16 (with intervening `\plmac@sep`s).

```
152   \edef\plmac@body{%
153     USE\plmac@sep
154     \plmac@tag\plmac@sep
155     \plmac@cleaned@macname
156   }%
```

**\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.

```
157    \let\plmac@hash=\relax
158    \plmac@argnum=\@ne
159    \loop
160      \ifnum\plmac@numargs<\plmac@argnum
161      \else
162        \edef\plmac@body{%
163          \plmac@body\plmac@sep\plmac@tag\plmac@sep
164          \plmac@hash\plmac@hash\number\plmac@argnum}%
165        \advance\plmac@argnum by \@ne
166    \repeat
167    \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`).

```
168    \expandafter\let\plmac@macname=\relax
169    \ifx\plmac@defarg\relax
170      \edef\plmac@define@command{%
171        \noexpand\plmac@command\plmac@starchar{\plmac@macname}%
172        [\plmac@numargs]{%
173          \noexpand\plmac@write@perl{\plmac@body}%
174        }%
175    }%
176    \else
177      \edef\plmac@define@command{%
178        \noexpand\plmac@command\plmac@starchar{\plmac@macname}%
179        [\plmac@numargs][\plmac@defarg]{%
180          \noexpand\plmac@write@perl{\plmac@body}%
181        }%
182    }%
183    \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`.

```
184    \expandafter\let\plmac@macname=\plmac@oldbody
185    \plmac@define@command
186    \plmac@next
187 }
```

### 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.

```
188 \def\perlnewenvironment{%
189    \let\plmac@command=\newcommand
190    \let\plmac@next=\plmac@end@environment
191    \@ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
192 }
```

```
193 \def\perlrenewenvironment{%
194    \let\plmac@command=\renewcommand
195    \let\plmac@next=\plmac@end@environment
196    \@ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
197 }
```

`\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 17 to understand the ba-

21

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`.

```
198 \def\plmac@newenvironment@i#1#2{%
199   \ifx#1*%
200     \def\plmac@starchar{*}%
201   \else
202     \def\plmac@starchar{}%
203   \fi
204   \def\plmac@envname{#2}%
205   \expandafter\def\expandafter\plmac@macname\expandafter{\csname#2\endcsname}%
206   \expandafter\let\expandafter\plmac@oldbody\plmac@macname\relax
207   \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
208     \expandafter\string\plmac@macname}%
209   \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%
210 }
```

`\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`.

```
211 \def\plmac@end@environment{%
212   \expandafter\def\expandafter\plmac@next\expandafter{\expandafter
213     \let\csname end\plmac@envname\endcsname=\plmac@end@macro
214     \let\plmac@next=\relax
215   }%
216   \def\plmac@macname{\plmac@end@macro}%
217   \expandafter\let\expandafter\plmac@oldbody\csname end\plmac@envname\endcsname
218   \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
219     \expandafter\string\plmac@macname}%
220   \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%
221 }
```

### 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`.

```
222 \def\perldo{%
223   \begingroup
224     \let\do\@makeother\dospecials
225     \catcode'\^^M=\active
226     \newlinechar'\^^M
227     \endlinechar='\^^M
228     \catcode'\{=1
229     \catcode'\}=2
230     \afterassignment\plmac@have@run@code
231     \global\plmac@perlcode
232 }
```

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

```
233 \def\plmac@have@run@code{%
234   \endgroup
235   \edef\plmac@run@code{%
236     \noexpand\plmac@write@perl{RUN\plmac@sep
237       \plmac@tag\plmac@sep
238       N/A\plmac@sep
239       \plmac@tag\plmac@sep
240       \the\plmac@perlcode
241     }%
242   }%
243   \plmac@run@code
244 }
```

### 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.

| Time | LATEX | | Perl |
|---|---|---|---|
| | Write `\plmac@tofile` | | |
| | Touch `\plmac@toflag` | → | Await `\plmac@toflag` |
| | | | Read `\plmac@tofile` |
| | | | Write `\plmac@fromfile` |
| | | | Delete `\plmac@toflag` |
| | | | Delete `\plmac@tofile` |
| | | | Delete `\plmac@doneflag` |
| | Await `\plmac@fromflag` | ← | Touch `\plmac@fromflag` |
| | Touch `\plmac@tofile` | → | Await `\plmac@tofile` |
| | | | Delete `\plmac@fromflag` |
| | Await `\plmac@doneflag` | ← | Touch `\plmac@doneflag` |
| | Read `\plmac@fromfile` | | |
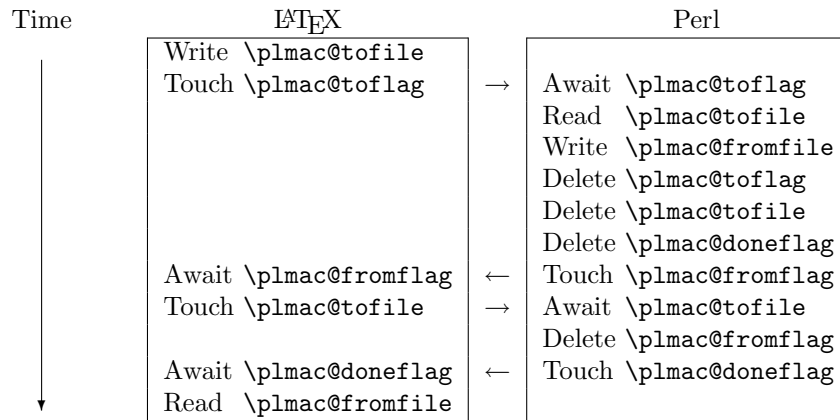
Figure 4: LATEX-to-Perl communication protocol

Although Figure 4 shows the read of `\plmac@fromfile` as the final step of the protocol, the file's contents are in fact valid as soon as LATEX detects that `\plmac@fromflag` exists. Deferring the read to the end, however, enables PerlTEX to support recursive macro invocations.

`\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.

As a performance optimization we `\input` a named pipe. This causes the `latex` process to relinquish the CPU until the `perltex` process writes data (always just a comment plus "`\endinput`") into the named pipe. On systems that don't support persistent named pipes (e.g., Microsoft Windows), `\plmac@pipe` is an ordinary file containing only a comment plus "`\endinput`". While reading that file is not guaranteed to relinquish the CPU, it should not hurt the performance or correctness of the communication protocol between LaTeX and Perl.

```
245 \newif\ifplmac@file@exists
```

```
246 \newcommand{\plmac@await@existence}[1]{%
247   \begin{lrbox}{\@tempboxa}%
248     \input\plmac@pipe
249   \end{lrbox}%
250   \loop
251     \IfFileExists{#1}%
252               {\plmac@file@existstrue}%
253               {\plmac@file@existsfalse}%
254     \ifplmac@file@exists
255     \else
256   \repeat
257 }
```

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

```
258 \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.

```
259 \newcommand{\plmac@write@perl}{%
260   \begingroup
261     \let\do\@makeother\dospecials
262     \catcode'\^^M=\active
263     \newlinechar'\^^M
264     \endlinechar='\^^M
265     \catcode'\{=1
266     \catcode'\}=2
267     \plmac@write@perl@i
268 }
```

`\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`.

```
269 \ifperl
270   \newcommand{\plmac@write@perl@i}[1]{%
```

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

```
271       \immediate\openout\plmac@outfile=\plmac@tofile\relax
272       \let\protect=\noexpand
273       \def\begin{\noexpand\begin}%
274       \def\end{\noexpand\end}%
275       \immediate\write\plmac@outfile{#1}%
276       \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.

```
277     \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`.

```
278       \immediate\openout\plmac@outfile=\plmac@toflag\relax
279       \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`.

```
280       \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 synchro-
nization 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`.

```
281       \immediate\openout\plmac@outfile=\plmac@tofile\relax
282       \immediate\closeout\plmac@outfile
283       \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.

```
284     \input\plmac@fromfile\relax
285   }
```

\plmac@write@perl@i  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 do-nothing macro and redefine it as "`\fbox{Perl\TeX}`" at the `\begin{document}`.

```
286 \else
287   \newcommand{\plmac@write@perl@i}[1]{\endgroup}
```

\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.

```
288   \gdef\plmac@show@placeholder#1#2\@empty{%
289     \ifx#1D\relax
290       \endgroup
291     \else
292       \endgroup
293       \fbox{Perl\TeX}%
294     \fi
295   }%

296   \AtBeginDocument{%
297     \renewcommand{\plmac@write@perl@i}[1]{%
298       \plmac@show@placeholder#1\@empty
299     }%
300   }
301 \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.

```
302 #! /usr/bin/env perl
303
304 ############################################################
```

```
305 # Prepare a LaTeX run for two-way communication with Perl #
306 # By Scott Pakin <scott+pt@pakin.org>                      #
307 ###########################################################
308
309 #------------------------------------------------------------------
310 # This is file 'perltex.pl',
311 # generated with the docstrip utility.
312 #
313 # The original source files were:
314 #
315 # perltex.dtx  (with options: 'perltex')
316 #
317 # This is a generated file.
318 #
319 # Copyright (C) 2010 Scott Pakin <scott+pt@pakin.org>
320 #
321 # This file may be distributed and/or modified under the conditions
322 # of the LaTeX Project Public License, either version 1.3c of this
323 # license or (at your option) any later version.  The latest
324 # version of this license is in:
325 #
326 #    http://www.latex-project.org/lppl.txt
327 #
328 # and version 1.3c or later is part of all distributions of LaTeX
329 # version 2006/05/20 or later.
330 #------------------------------------------------------------------
331
```

### 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.

```
332 sub top_level_eval ($)
333 {
334     return eval $_[0];
335 }
```

### 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.

```
336 use Safe;
```

```
337 use Opcode;
338 use Getopt::Long;
339 use Pod::Usage;
340 use File::Basename;
341 use Fcntl;
342 use POSIX;
343 use File::Spec;
344 use IO::Handle;
345 use warnings;
346 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  
$usepipe

`$latexprog` is the name of the LaTeX executable (e.g., "`latex`"). If `$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`. `$usepipe` is 1 if `perltex.pl` should attempt to use a named pipe for communicating with `latex` or 0 if an ordinary file should be used instead.

```
347 my $latexprog;
348 my $runsafely = 1;
349 my @permittedops;
350 my $usepipe = 1;
```

**Filename variables**

$progname  
$jobname  
$toperl  
$fromperl  
$toflag  
$fromflag  
$doneflag  
$logfile  
$pipe

`$progname` is the run-time name of the `perltex.pl` program. `$jobname` is the base name of the user's `.tex` file, which defaults to the TeX default of `texput`. `$toperl` defines the filename used for LaTeX-to-Perl communication. `$fromperl` defines the filename used for Perl-to-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`. `$logfile` is the name of a log file to which `perltex.pl` writes verbose execution information. `$pipe` is the name of a Unix named pipe (or ordinary file on operating systems that lack support for persistent named pipes or in the case that `$usepipe` is set to 0) used to convince the `latex` process to yield control of the CPU.

```
351 my $progname = basename $0;
352 my $jobname = "texput";
353 my $toperl;
```

```
354 my $fromperl;
355 my $toflag;
356 my $fromflag;
357 my $doneflag;
358 my $logfile;
359 my $pipe;
```

### Other global variables

@latexcmdline    `@latexcmdline` is the command line to pass to the LATEX executable. `$styfile` is
$styfile    the string `noperltex.sty` if `perltex.pl` is run with `--makesty`, otherwise unde-
@macroexpansions    fined. `@macroexpansions` is a list of PerlTEX macro expansions in the order they
$sandbox    were encountered. It is used for creating a `noperltex.sty` file when `--makesty`
$sandbox_eval    is specified. `$sandbox` is a secure sandbox in which to run code that appeared
$latexpid    in the LATEX document. `$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.

```
360 my @latexcmdline;
361 my $styfile;
362 my @macroexpansions;
363 my $sandbox = new Safe;
364 my $sandbox_eval;
365 my $latexpid;
```

$pipestring    `$pipestring` is a constant string to write to the `$pipe` named pipe (or file) at each
LATEX synchronization point. Its particular definition is really a bug workaround
for XƎTEX. The current version of XƎTEX reads the first few bytes of a file to
determine the character encoding (UTF-8 or UTF-16, big-endian or little-endian)
then attempts to rewind the file pointer. Because pipes can't be rewound, the effect
is that the first two bytes of `$pipe` are discarded and the rest are input. Hence,
the "`\endinput`" used in prior versions of PerlTEX inserted a spurious "`ndinput`"
into the author's document. We therefore define `$pipestring` such that it will
not interfere with the document even if the first few bytes are discarded.

```
366 my $pipestring = "\%\%\%\%\% Generated by $progname\n\\endinput\n";
```

### 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 con-
tents 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.

```
367 $latexprog = $ENV{"PERLTEX"} || "latex";
368 Getopt::Long::Configure("require_order", "pass_through");
```

```
369 GetOptions("help"        => sub {pod2usage(-verbose => 1)},
370           "latex=s"     => \$latexprog,
371           "safe!"       => \$runsafely,
```

The following two options are undocumented because the defaults should always suffice. We're not yet removing these options, however, in case they turn out to be useful for diagnostic purposes.

```
372           "pipe!"       => \$usepipe,
373           "synctext=s"  => \$pipestring,

374           "makesty"     => sub {$styfile = "noperltex.sty"},
375           "permit=s"    => \@permittedops) || pod2usage(2);
```

### Preparing a LATEX 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.

```
376 @latexcmdline = @ARGV;
377 my $firstcmd = 0;
378 for ($firstcmd=0; $firstcmd<=$#latexcmdline; $firstcmd++) {
379     my $option = $latexcmdline[$firstcmd];
380     next if substr($option, 0, 1) eq "-";
381     if (substr ($option, 0, 1) ne "\\") {
382         $jobname = basename $option, ".tex" ;
383         $latexcmdline[$firstcmd] = "\\input $option";
384     }
385     last;
386 }
387 push @latexcmdline, "" if $#latexcmdline==-1;
```

$separator  To avoid conflicts with the code and parameters passed to Perl from LATEX (see Figure 1 on page 16 and Figure 2 on page 16) we define a separator string, $separator, containing 20 random uppercase letters.

```
388 my $separator = "";
389 foreach (1 .. 20) {
390     $separator .= chr(ord("A") + rand(26));
391 }
```

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

```
392 $toperl = $jobname . ".topl";
393 $fromperl = $jobname . ".frpl";
394 $toflag = $jobname . ".tfpl";
395 $fromflag = $jobname . ".ffpl";
396 $doneflag = $jobname . ".dfpl";
397 $logfile = $jobname . ".lgpl";
398 $pipe = $jobname . ".pipe";
```

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.

```
399 $latexcmdline[$firstcmd] =
400     sprintf '\makeatletter' . '\def%s{%s}' x 7 . '\makeatother%s',
401     '\plmac@tag', $separator,
402     '\plmac@tofile', $toperl,
403     '\plmac@fromfile', $fromperl,
404     '\plmac@toflag', $toflag,
405     '\plmac@fromflag', $fromflag,
406     '\plmac@doneflag', $doneflag,
407     '\plmac@pipe', $pipe,
408     $latexcmdline[$firstcmd];
```

### 3.2.6   Increasing PerlTEX's robustness

```
409 $toperl = File::Spec->rel2abs($toperl);
410 $fromperl = File::Spec->rel2abs($fromperl);
411 $toflag = File::Spec->rel2abs($toflag);
412 $fromflag = File::Spec->rel2abs($fromflag);
413 $doneflag = File::Spec->rel2abs($doneflag);
414 $logfile = File::Spec->rel2abs($logfile);
415 $pipe = File::Spec->rel2abs($pipe);
```

perltex.pl may hang if `latex` exits right before the final pipe communication. We therefore define a simple SIGALRM handler that lets `perltex.pl` exit after a given length of time has elapsed.

```
416 $SIG{"ALRM"} = sub {
417     undef $latexpid;
418     exit 0;
419 };
```

To prevent Perl from aborting with a "`Broken pipe`" error message if `latex` exits during the final pipe communication we tell Perl to ignore SIGPIPE errors. `latex`'s exiting will be caught via other means (the preceding SIGALRM handler or the following call to `waitpid`).

```
420 $SIG{"PIPE"} = "IGNORE";
```

delete_files   On some operating systems and some filesystems, deleting a file may not cause the file to disappear immediately. Because PerlTEX synchronizes Perl and LATEX via the filesystem it is critical that file deletions be performed when requested. We therefore define a `delete_files` subroutine that waits until each file named in the argument list is truly deleted.

```
421 sub delete_files (@)
422 {
423     foreach my $filename (@_) {
424         unlink $filename;
425         while (-e $filename) {
426             unlink $filename;
427             sleep 0;
```

```
428            }
429        }
430 }
```

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

```
431 sub awaitexists ($)
432 {
433     while (!-e $_[0]) {
434         sleep 0;
435         if (waitpid($latexpid, &WNOHANG)==-1) {
436             delete_files($toperl, $fromperl, $toflag,
437                          $fromflag, $doneflag, $pipe);
438             undef $latexpid;
439             exit 0;
440         }
441     }
442 }
```

### 3.2.7   Launching LATEX

We start by deleting the `$toperl`, `$fromperl`, `$toflag`, `$fromflag`, `$doneflag`, and `$pipe` files, in case any of these were left over from a previous (aborted) run. We also create a log file (`$logfile`), a named pipe (`$pipe`)—or a file containing only `\endinput` if we can't create a named pipe—and, if `$styfile` is defined, a LATEX 2$_\varepsilon$ style file. 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.

```
443 delete_files($toperl, $fromperl, $toflag, $fromflag, $doneflag, $pipe);
444 open (LOGFILE, ">$logfile") || die "open(\"$logfile\"): $!\n";
445 autoflush LOGFILE 1;
446 if (defined $styfile) {
447     open (STYFILE, ">$styfile") || die "open(\"$styfile\"): $!\n";
448 }

449 if (!$usepipe || !eval {mkfifo($pipe, 0600)}) {
450     sysopen PIPE, $pipe, O_WRONLY|O_CREAT, 0755;
451     autoflush PIPE 1;
452     print PIPE $pipestring;
453     close PIPE;
454     $usepipe = 0;
455 }

456 defined ($latexpid = fork) || die "fork: $!\n";
457 unshift @latexcmdline, $latexprog;
458 if (!$latexpid) {
459     exec {$latexcmdline[0]} @latexcmdline;
460     die "exec('@latexcmdline'): $!\n";
461 }
```

33

### 3.2.8 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.

```
462 if ($runsafely) {
463     @permittedops=(":browse") if $#permittedops==-1;
464     $sandbox->permit_only (@permittedops);
465     $sandbox_eval = sub {$sandbox->reval($_[0])};
466 }
467 else {
468     $sandbox_eval = \&top_level_eval;
469 }
```

### 3.2.9 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 24.

```
470 while (1) {
```

$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.

```
471     awaitexists($toflag);
472     my $entirefile;
473     {
474         local $/ = undef;
475         open (TOPERL, "<$toperl") || die "open($toperl): $!\n";
476         $entirefile = <TOPERL>;
477         close TOPERL;
478     }
```

$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
@otherstuff    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.

```
479     $entirefile =~ s/\r//g;
480     my ($optag, $macroname, @otherstuff) =
481         map {chomp; $_} split "$separator\n", $entirefile;
```

34

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.

```
482    $macroname =~ s/^[^A-Za-z]+//;
483    $macroname =~ s/\W/_/g;
484    $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 "\'".

```
485    if ($optag eq "USE") {
486      foreach (@otherstuff) {
487          s/\\/\\\\/g;
488          s/\'/\\\'/g;
489          $_ = "'$_'";
490      }
491    }
```

**$perlcode**  There are three possible values that can be assigned to `$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.

```
492    my $perlcode;
493    if ($optag eq "DEF") {
494        $perlcode =
495            sprintf "sub %s {%s}\n",
496            $macroname, $otherstuff[0];
```

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

⇓

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

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

35

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

$$\Downarrow$$

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

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

```
497    }
498    elsif ($optag eq "USE") {
499        $perlcode = sprintf "%s (%s);\n", $macroname, join(", ", @otherstuff);
500    }
501    elsif ($optag eq "RUN") {
502        $perlcode = $otherstuff[0];
503    }
504    else {
505        die "${progname}: Internal error -- unexpected operation tag \"$optag\"\n";
506    }
```

Log what we're about to evaluate.

```
507    print LOGFILE "#" x 31, " PERL CODE ", "#" x 32, "\n";
508    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 $2_\varepsilon$'s `\PackageError` macro to return a suitable error message. We produce one error message for sandbox policy violations (detected by the error message, `$@`, containing the string "`trapped by`") and a different error message for all other errors caused by executing the user's code. For clarity of reading both warning and error messages, we elide the string "`at (eval ⟨number⟩) line ⟨number⟩`". Once `$result` is defined—as either the resulting LaTeX code or as a `\PackageError`—we store it in `@macroexpansions` in preparation for writing it to `noperltex.sty` (when `perltex.pl` is run with `--makesty`).

```
509    undef $_;
510    my $result;
511    {
512        my $warningmsg;
513        local $SIG{__WARN__} =
514            sub {chomp ($warningmsg=$_[0]); return 0};
515        $result = $sandbox_eval->($perlcode);
516        if (defined $warningmsg) {
517            $warningmsg =~ s/at \(eval \d+\) line \d+\W+//;
518            print LOGFILE "# ===> $warningmsg\n\n";
519        }
520    }
521    $result = "" if !$result || $optag eq "RUN";
```

36

```
522    if ($@) {
523        my $msg = $@;
524        $msg =~ s/at \(eval \d+\) line \d+\W+//;
525        $msg =~ s/\n/\\MessageBreak\n/g;
526        $msg =~ s/\s+/ /;
527        $result = "\\PackageError{perltex}{$msg}";
528        my @helpstring;
529        if ($msg =~ /\btrapped by\b/) {
530            @helpstring =
531                ("The preceding error message comes from Perl.  Apparently,",
532                 "the Perl code you tried to execute attempted to perform an",
533                 "'unsafe' operation.  If you trust the Perl code (e.g., if",
534                 "you wrote it) then you can invoke perltex with the --nosafe",
535                 "option to allow arbitrary Perl code to execute.",
536                 "Alternatively, you can selectively enable Perl features",
537                 "using perltex's --permit option.  Don't do this if you don't",
538                 "trust the Perl code, however; malicious Perl code can do a",
539                 "world of harm to your computer system.");
540        }
541        else {
542            @helpstring =
543                ("The preceding error message comes from Perl.  Apparently,",
544                 "there's a bug in your Perl code.  You'll need to sort that",
545                 "out in your document and re-run perltex.");
546        }
547        my $helpstring = join ("\\MessageBreak\n", @helpstring);
548        $helpstring =~ s/\.  /.\\space\\space /g;
549        $result .= "{$helpstring}";
550    }
551    push @macroexpansions, $result if defined $styfile && $optag eq "USE";
```

Log the resulting LATEX code.

```
552    print LOGFILE "%" x 30, " LATEX RESULT ", "%" x 30, "\n";
553    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.

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

Continuing the protocol described in Figure 4 on page 24 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. As a performance optimization, we also write \endinput into $pipe to wake up the latex process.

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

558    delete_files($toflag, $toperl, $doneflag);

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

```
561    if (open (PIPE, ">$pipe")) {
562        autoflush PIPE 1;
563        print PIPE $pipestring;
564        close PIPE;
565    }
```

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.

```
566    awaitexists($toperl);
567    delete_files($fromflag);
568    open (DONEFLAG, ">$doneflag") || die "open($doneflag): $!\n";
569    close DONEFLAG;
```

Again, we awaken the latex process, which is blocked on $pipe. If writing to the pipe takes more than one second we assume that latex has exited and trigger the SIGALRM handler (page 32).

```
570    alarm 1;
571    if (open (PIPE, ">$pipe")) {
572        autoflush PIPE 1;
573        print PIPE $pipestring;
574        close PIPE;
575    }
576    alarm 0;
577 }
```

### 3.2.10   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.

```
578 END {
579    close LOGFILE;
580    if (defined $latexpid) {
581        kill (9, $latexpid);
582        exit 1;
583    }
584
585    if (defined $styfile) {
```

This is the big moment for the --makesty option. We've accumulated the output from each PerlTeX macro invocation into @macroexpansions, and now we need to produce a noperltex.sty file. We start by generating a boilerplate header in which we set up the package and load both perltex and filecontents.

```
586        print STYFILE <<"STYFILEHEADER1";
587 \\NeedsTeXFormat{LaTeX2e}[1999/12/01]
588 \\ProvidesPackage{noperltex}
589    [2007/09/29 v1.4 Perl-free version of PerlTeX specific to $jobname.tex]
590 STYFILEHEADER1
591        ;
592        print STYFILE <<'STYFILEHEADER2';
```

38

```
593 \RequirePackage{filecontents}
594
595 % Suppress the "Document must be compiled using perltex" error from perltex.
596 \let\noperltex@PackageError=\PackageError
597 \renewcommand{\PackageError}[3]{}
598 \RequirePackage{perltex}
599 \let\PackageError=\noperltex@PackageError
600
```

\plmac@macro@invocation@num   noperltex.sty works by redefining the \plmac@show@placeholder macro, which
\plmac@show@placeholder       normally outputs a framed "PerlTeX" when perltex.pl isn't running, changing
it to input noperltex-⟨*number*⟩.tex instead (where ⟨*number*⟩ is the contents
of the \plmac@macro@invocation@num counter). Each noperltex-⟨*number*⟩.tex
file contains the output from a single invocation of a PerlTeX-defined macro.

```
601 % Modify \plmac@show@placeholder to input the next noperltex-*.tex file
602 % each time a PerlTeX-defined macro is invoked.
603 \newcount\plmac@macro@invocation@num
604 \gdef\plmac@show@placeholder#1#2\@empty{%
605   \ifx#1U\relax
606     \endgroup
607     \advance\plmac@macro@invocation@num by 1\relax
608     \global\plmac@macro@invocation@num=\plmac@macro@invocation@num
609     \input{noperltex-\the\plmac@macro@invocation@num.tex}%
610   \else
611     \endgroup
612   \fi
613 }
614 STYFILEHEADER2
615           ;
```

Finally, we need to have noperltex.sty generate each of the
noperltex-⟨*number*⟩.tex files. For each element of @macroexpansions we
use one filecontents environment to write the macro expansion verbatim to a
file.

```
616        foreach my $e (0 .. $#macroexpansions) {
617            print STYFILE "\n";
618            printf STYFILE "%% Invocation #%d\n", 1+$e;
619                printf STYFILE "\\begin{filecontents}{noperltex-%d.tex}\n", 1+$e;
620            print STYFILE $macroexpansions[$e], "\\endinput\n";
621            print STYFILE "\\end{filecontents}\n";
622        }
623        print STYFILE "\\endinput\n";
624        close STYFILE;
625    }
626
627    exit 0;
628 }
629
630 __END__
```

### 3.2.11 `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 2.3. 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

Copyright © 2010 Scott Pakin `<scott+pt@pakin.org>`

These files 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.

# Acknowledgments

Thanks to Andrew Mertz for writing the first draft of the code that produces the PerlTeX-free `noperltex.sty` style file and for testing the final draft; to Andrei Alexandrescu for providing a few bug fixes; to Nick Andrewes for identifying and helping diagnose a problem running PerlTeX with X‚ÄÖTeX and to Jonathan Kew for suggesting a workaround; and to Linus Källberg for reporting and helping diagnose some problems with running PerlTeX on Windows. Also, thanks to the many people who have sent me fan mail or submitted bug reports, documentation corrections, or feature requests. (The `\perldo` macro and the `--makesty` option were particularly popular requests.)

# Change History

# 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.