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 2ε 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

LaTeX 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 \texttt{substr} function and Perl\LaTeX makes it easy to export this to \LaTeX:

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

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

\verbatimcode{\newcommand\{str\}{superlative}
A sample substring of ‘‘\texttt{str}’’ is ‘‘\texttt{substr\{str\}\{2\}\{4\}}’’.

\verbatimcode{\text\{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:

\verbatimcode{perlnewcommand\{hilbertmatrix\}[1]{
 \my \$result = ‘‘
 \[\]
 \renewcommand\{arraystretch\}{1.3}
 ‘‘
 \$result .= ‘‘\begin{array}{' . 'c' x $[0] . '}
 \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) . ‘
 \}
 \$result .= ‘‘\end{array}
 \‘‘
 \}
 \hilbertmatrix{20}

\verbatimcode{2}
In addition to \perlnewcommand and \perlnorenewcommand, Perl\TeX{} supports \perlnewenvironment and \perlnorenewenvironment 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 Perl\TeX{}:

\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 .= \newcommand{\rownum}{\thesrow\addtocounter{ssrow}{1}}\n";
  $tabular .= \setcounter{ssrow}{1}\n";
  $header = "A";
  $cols = $_[0];
  if ($cols == 1) {
    $tabular .= "\begin{tabular}{|c|}\hline\n";
    $tabular .= $header ++ " \hline\n";
    $tabular .= "$\\" \hline\n";
  } else {
    $tabular .= "$\&\hline\n";
  }
  $tabular .= "$\n";
}
\begin{tabular}{l}
\rownum & 1 & 8 & 10 & 15 \\
\rownum & 12 & 13 & 3 & 6 \\
\rownum & 7 & 2 & 16 & 9 \\
\rownum & 14 & 11 & 5 & 4
\end{tabular}

2 Usage

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

2.1 Defining and redefining Perl macros

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

A Perl\TeX-defined macro or environments is converted to a Perl subroutine named after the macro/environment but beginning with \texttt{latex_}. For example, a Perl\TeX-defined \LaTeX\ macro called \texttt{\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, Perl\TeX executes the Perl code within a secure sandbox. This means that potentially harmful Perl operations, such as \texttt{unlink}, \texttt{rmdir}, and \texttt{system} will result in a runtime 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 Perl\TeX 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 \texttt{\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
```

\[
\downarrow
\]

\[x\text{ was set to } 123. x\text{ was set to } 456. x\text{ was set to } 789.\]

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

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

An invocation of “\texttt{\verbit{\TeX}}” would therefore typeset the expansion of “\TeX”, namely “T\texttt{kern} \texttt{-1.667em}\texttt{lower} .5ex\texttt{hbox} \{E}\texttt{kern} \texttt{-1.25emX}\texttt{spacefactor} \texttt{\_m}”, which might be a bit unexpected. The solution is to use \texttt{\noexpand\verbit{\noexpand\TeX}} ⇒ \TeX. “Robust” macros as well as \texttt{\begin} and \texttt{\end} are implicitly preceded by \texttt{\noexpand}.

## 2.2 Making perltex.pl optional

Normally, perltex.sty issues a Document must be compiled using perltex error if a document specifies \texttt{\usepackage{perltex}} but is not compiled using perltex.pl. However, sometimes Perl\TeX may be needed merely to enhance a
document’s formatting without being mandatory for compiling the document. For such cases, the \texttt{optional} package option instructs \texttt{perltex.sty} only to note that Document was compiled without using the perltex script without aborting the compilation. The author can then use the \texttt{\ifperl} macro to test if \texttt{perltex.pl} is being used and, if not, provide alternative definitions for macros and environments defined with \texttt{\perlnewcommand} and \texttt{\perlnewenvironment}.

See Section \ref{sec:optional} for a large Perl\TeX{} example that uses \texttt{optional} and \texttt{\ifperl} to define an environment one way if \texttt{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 \texttt{perltex.sty} is not even installed.

\section{Invoking perltex.pl}

The following pages reproduce the \texttt{perltex.pl} program documentation. Key parts of the documentation are excerpted when \texttt{perltex.pl} is invoked with the \texttt{--help} option. The various Perl \texttt{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 \texttt{perltex.pl}:

\begin{verbatim}
    pod2man --center=" " --release=" " perltex.pl > perltex.1
\end{verbatim}
NAME
perltex — enable \LaTeX\ macros to be defined in terms of Perl code

SYNOPSIS
perltex [--help] [--latex=program] [--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. \texttt{perltex} is the tool that enables a symbiosis between the two systems. All a user needs to do is compile a \LaTeX\ document using \texttt{perltex} instead of \texttt{latex}. (\texttt{perltex} is actually a wrapper for \texttt{latex}, so no \texttt{latex} functionality is lost.) If the document includes a \texttt{\usepackage{perltex}} in its preamble, then \texttt{\perlnewcommand} and \texttt{\perlrnewcommand} macros will be made available. These behave just like \LaTeX’s \texttt{\newcommand} and \texttt{\renewcommand} except that the macro body contains Perl code instead of \LaTeX\ code.

OPTIONS
\texttt{perltex} accepts the following command-line options:

--help

Display basic usage information.

--latex=program

Specify a program to use instead of \texttt{latex}. For example, \texttt{--latex=pdflatex} would typeset the given document using \texttt{pdflatex} instead of ordinary \texttt{latex}.

--[no]safe

Enable or disable sandboxing. With the default of \texttt{--safe}, \texttt{perltex} executes the code from a \texttt{\perlnewcommand} or \texttt{\perlrnewcommand} macro within a protected environment that prohibits “unsafe” operations such as accessing files or executing external programs. Specifying \texttt{--nosafe} gives the \LaTeX\ document \textit{carte blanche} to execute any arbitrary Perl code, including that which can harm the user’s files. See \textit{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 \texttt{\usepackage{perltex}} line with \texttt{\usepackage{noperltex}} produces the same output but does not require Perl\TeX{}, making the document suitable for distribution to people who do not have Perl\TeX{} installed. The disadvantage is that noperltex.sty is specific to the document that produced it. Any changes to the document’s Perl\TEX{} 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 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 \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, perltext 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 Perl\TeX{} macro invocation

NOTES
perltext’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 Perl\TeX{}.

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 Perl\TeX{} macro \texttt{\textbackslash 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. \texttt{\textbackslash dolinkwords} first replaces all sequences of the form \assoc{⟨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, \texttt{\textbackslash dolinkwords} replaces the dummy sequences with the corresponding \LaTeX{} text and returns the result.

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

As an added bonus, \texttt{\textbackslash ifperl\ldots\textbackslash else\ldots\textbackslash fi} is used to surround the definition of the \texttt{\textbackslash 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 perl\texttt{\textbackslash tex\textbackslash sty} is loaded with the optional option to indicate that the document can compile without perltex.pl. However, the user still needs perl\texttt{\textbackslash tex\textbackslash sty} to avoid getting a File ‘perl\texttt{\textbackslash tex\textbackslash sty}’ not found error from \LaTeX{}. To produce a document that can compile even without perl\texttt{\textbackslash tex\textbackslash sty} installed, replace the \texttt{\usepackage[optional]{perl\texttt{\textbackslash tex}}} line with
the following \LaTeX code:

\IfFileExists{perl\text{tex}.sty}
\{\usepackage[optional]{perl\text{tex}}\}
\{\newif\ifperl\}

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

tex perl\text{tex}.\text{ins}

In the following listing, line numbers are suffixed with “X” to distinguish them from line numbers associated with Perl\LaTeX's source code.

1X \documentclass{article}
2X \usepackage[optional]{perl\text{tex}}
3X \usepackage{environ}
4X \usepackage{hyperref}
5X
6X \ifperl
7X \perlnewcommand\{\dolinkwords\}[3]{
8X \texttt{# Preprocess our arguments.}\n9X \$url = \$_[0];\n10X \$url =~ s/\%s/\%s/g;\n11X \%stopwords = map \{lc \$_ => 1\} split " ", \$_[1];\n12X \$stopwords(""") = 1;\n13X \$text = \$_[2];\n14X \}
15X
16X \texttt{# Replace \LaTeX code in the text with placeholders.}\n17X \$placeholder = "ABC\text{zyz}123";\n18X \%subs = ();\n19X \$replace = sub \{\$subs{$\text{placeholder}$} = \$_[0]; \$\text{placeholder}++;\}\n20X \$text =~ s/\begin\{[a-z]\}\+$replace->($&)/gse;\n21X \$text =~ s/\[a-z]/$replace->($&)/gse;\n22X
23X \texttt{# Hyperlink each word that's not in the stop list.}\n24X \$newtext = "";\n25X foreach \$\text{word} (split /((?<=-[-A\text{a}-z]a-z]+b)/i, \$text) {\n26X \$l\text{\text{cword}} = lc \$\text{\text{word}};\n27X \texttt{if (defined \$stopwords{$\text{l\text{\text{cword}}}$} || $\text{l\text{\text{cword}}} =~ /[^a-z]/) \{\n28X \$newtext . -= \$\text{\text{word}};\n29X \}}\n30X \texttt{else \{\n31X \$newtext . -= sprintf "<a href="\$url">\%s\</a">", $\text{l\text{\text{cword}}}, $\text{\text{\text{word}};\n32X \}}\n33X \}}
# Restore original text from placeholders and return the new text.
while (($tag, $orig) = each %substs) {
    $newtext =~ s/\Q$tag\E/$orig/gs;
}
return $newtext;
\NewEnviron{linkwords}[2][]{{\dolinkwords{#2}{#1}{\BODY}}}
\else
\newenvironment{linkwords}[2][]{}
\fi
\begin{document}
\newcommand\stopwords{a an the of in am and or but i we me you us them}
\begin{linkwords}\stopwords{http://www.google.com/search?q=define:\%s}
\begin{verse}
I'm very good at integral and differential calculus; \\\nI know the scientific names of beings animalculous: \\\nIn short, in matters vegetable, animal, and mineral, \\\nI am the very model of a modern Major-General.
\end{verse}
\end{linkwords}
\end{document}

### 3 Implementation

Users interested only in using Perl\TeX{} can skip Section 3, which presents the complete Perl\TeX{} source code. This section should be of interest primarily to those who wish to extend Perl\TeX{} 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 Perl\TeX{}, and Section 3.2 presents the source code for perltex.pl, the Perl side of Perl\TeX{}. In toto, Perl\TeX{} 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 \LaTeX{} 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@perl0i`.

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

The `optional` package option lets an author specify that the document can be built successfully even without Perl\LaTeX{}. Typically, this means that the document uses `\ifperl` to help define reduced-functionality equivalents of any document-defined Perl\LaTeX{} 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.

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

Perl\LaTeX{} 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>
<thead>
<tr>
<th>Variable</th>
<th>Purpose</th>
<th>perltex.pl assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>\plmac@tag</td>
<td>\plmac@tofile field separator</td>
<td>(20 random letters)</td>
</tr>
<tr>
<td>\plmac@tofile</td>
<td>\LaTeX \rightarrow \text{Perl communication}</td>
<td>\jobname.topl</td>
</tr>
<tr>
<td>\plmac@fromfile</td>
<td>\text{Perl} \rightarrow \LaTeX communication</td>
<td>\jobname.frpl</td>
</tr>
<tr>
<td>\plmac@toflag</td>
<td>\plmac@tofile synchronization</td>
<td>\jobname.tfpl</td>
</tr>
<tr>
<td>\plmac@fromflag</td>
<td>\plmac@fromfile synchronization</td>
<td>\jobname.ffpl</td>
</tr>
<tr>
<td>\plmac@doneflag</td>
<td>\plmac@fromflag synchronization</td>
<td>\jobname.dfpl</td>
</tr>
</tbody>
</table>

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 \texttt{\texttt{perl|re|newcommand}} and \texttt{\texttt{perl|re|newenvironment}}.

\begin{verbatim}
\ifperl
\perltrue
\perlfalse
\ifundefined{plmac@tag}{\perlfalse}{\}
\ifundefined{plmac@tofile}{\ perlfalse}{\}
\ifundefined{plmac@fromfile}{\ perlfalse}{\}
\ifundefined{plmac@toflag}{\ perlfalse}{\}
\ifundefined{plmac@fromflag}{\ perlfalse}{\}
\ifundefined{plmac@doneflag}{\ perlfalse}{\}
\ifundefined{plmac@pipe}{\ perlfalse}{\}
\ifperl
\else
\end{verbatim}

\ifplmac@required
\PackageError{perltex}{Document must be compiled using perltex}
{Instead of compiling your document directly with \texttt{latex}, you need}
{to use the perltex script. \texttt{space perltex} sets up
a variety of macros needed for communication between \LaTeX\ and \texttt{perl}.}
\else
\begroup
\obeyspaces
\typeout{perltex: Document was compiled without using the perltex script;}
\typeout{ it may not print as desired.}
\egroup
\fi
\fi
3.1.2 Defining Perl macros

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

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

1. The user invokes \textbackslash \textnormal{perl}[\textnormal{re}]newcommand, which stores \textbackslash [\textnormal{re}]newcommand in \textbackslash \textnormal{plmac}@command. The \textbackslash [\textnormal{re}]newcommand macro then invokes \textbackslash \textnormal{plmac}@newcommand@i with a first argument of “*” for \textbackslash [\textnormal{re}]newcommand* or “!” for ordinary \textbackslash [\textnormal{re}]newcommand.

2. \textbackslash \textnormal{plmac}@newcommand@i defines \textbackslash \textnormal{plmac}@starchar as “*” if it was passed a “*” or ⟨empty⟩ if it was passed a “!” \text{. It then stores the name of the user’s macro in } \textbackslash \textnormal{plmac}@macname, \text{ a writeable version of the name in } \textbackslash \textnormal{plmac}@cleaned@macname, \text{ and the macro’s previous definition (needed by } \textbackslash \textnormal{perlrenewcommand} \text{ in } \textbackslash \textnormal{plmac}@oldbody. Finally, } \textbackslash \textnormal{plmac}@newcommand@i \text{ invokes } \textbackslash \textnormal{plmac}@newcommand@ii.

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

4. \textbackslash \textnormal{plmac}@newcommand@ii@opt defines \textbackslash \textnormal{plmac}@defarg as the default value of the optional argument. \textbackslash \textnormal{plmac}@newcommand@ii@no@opt defines it as ⟨empty⟩. Both functions then call \textbackslash \textnormal{plmac}@haveargs.

5. \textbackslash \textnormal{plmac}@haveargs stores the user’s macro body (written in Perl) verbatim in \textbackslash \textnormal{plmac}@perlcode. \textbackslash \textnormal{plmac}@haveargs then invokes \textbackslash \textnormal{plmac}@havecode.

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

7. \textbackslash \textnormal{plmac}@havecode invokes \textbackslash \textnormal{newcommand} or \textbackslash \textnormal{renewcommand}, as appropriate, defining the user’s macro as a call to \textbackslash \textnormal{plmac}@write@perl. An invocation of the user’s \LaTeX \text{ macro causes } \textbackslash \textnormal{plmac}@write@perl to pass the information shown in Figure 2 to perl\tex\text{.pl.}
Figure 1: Data written to `\plmac@tofile` to define a Perl subroutine

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

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

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

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

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

\ plmac@newcommand@i 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@macro 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.

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

\ plmac@newcommand@ii \ plmac@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@ii@opt if the first argument is to be optional or \ plmac@newcommand@iid@no@opt if all arguments are to be mandatory.

\ def\ plmac@newcommand@ii@#1{\%\\def\ plmac@newcommand@ii@#1{\%\def\ plmac@newcommand@ii@#1{\%
Only one of these two macros is executed per invocation of \perl[re]newcommand, depending on whether or not the first argument of the user's macro is an optional argument. \plmac@newcommand@iii@opt is invoked if the argument is optional. It defines \plmac@defarg to the default value of the optional argument. \plmac@newcommand@iii@no@opt is invoked if all arguments are mandatory. It defines \plmac@defarg as \relax. Both \plmac@newcommand@iii@opt and \plmac@newcommand@iii@no@opt then invoke \plmac@haveargs.

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

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

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

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

\newtoks\plmac@perlcode
\def\plmac@haveargs{%  
  \begingroup  
  \let\do@makeother\dospecials  
  \catcode'\^^M=\active  
  \newlinechar'\^^M  
  \endlinechar'\^^M  
  \catcode'\*=1  
  \catcode'\}=2  
  \afterassignment\plmac@havecode
  \global\plmac@perlcode
}
Control is 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 3⟩}”. This is easier said than done because the number of arguments in the user’s macro is not known statically, yet we need to iterate over however many arguments there are. Because of this complexity, we will explain \plmac@perlcode piece-by-piece.

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

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

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

\newcount\plmac@argnum

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

\def\plmac@havecode{%
  \endgroup
}

\plmac@define@sub
We invoke \plmac@write@perl to define a Perl subroutine named after \plmac@cleaned@macname. \plmac@define@sub sends Perl the information shown in Figure 1 on page 16.

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

\plmac@body
The rest of \plmac@havecode is preparation for defining the user’s macro. \LaTeX{}’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@seps).

\edef\plmac@body{%
  USE\plmac@sep
  \plmac@tag\plmac@sep
  \plmac@cleaned@macname
}%
Now, for each argument \texttt{#1}, \texttt{#2}, \ldots, \texttt{#\plmac@numargs} we append a \texttt{\plmac@tag} plus the argument to \texttt{\plmac@body} (as always, with a \texttt{\plmac@sep} after each item). This requires more trickery, as \LaTeX{} 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 \Hypertext{} and probably other \TeX{}-based systems as well), is to \texttt{\let} \texttt{\plmac@hash} to \texttt{\relax}. This makes \texttt{\plmac@hash} unexpandable, and because it's not a "#", \TeX{} doesn't complain. After \texttt{\plmac@body} has been extended to include \texttt{\plmac@hash1}, \texttt{\plmac@hash2}, \ldots, \texttt{\plmac@hash\plmac@numargs}, we then \texttt{\let}-bind \texttt{\plmac@hash} to \texttt{##}, which \TeX{} lets us do because we're within a macro definition (\texttt{\plmac@havecode}). \texttt{\plmac@body} will then contain \texttt{#1}, \texttt{#2}, \ldots, \texttt{#\plmac@numargs}, as desired.

\begin{verbatim}
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=##%
\end{verbatim}

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

\begin{verbatim}
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  }%
174 \else
175  \edef\plmac@define@command{%
176    \noexpand\plmac@command\plmac@starchar{\plmac@macname}%
177    \{\plmac@numargs\ \plmac@defarg\}%
178  }%
179 \fi
\end{verbatim}

The final steps are to restore the previous definition of the user's macro—we had set it to \texttt{\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 \perlrerecommand but processes the end-environment
code for \perlnewenvironment and \perlrereenvironment.

\begin{verbatim}
184 \expandafter\let\plmac@macname=\plmac@oldbody
185 \plmac@define@command
186 \plmac@next
187 \}
\end{verbatim}

3.1.3 Defining Perl environments

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

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

\begin{verbatim}
\plmac@newenvironment@i
\plmac@starchar
\plmac@envname
\plmac@macname
\plmac@oldbody
\plmac@cleaned@macname
\end{verbatim}

The \plmac@newenvironment@i macro is analogous to \plmac@newcommand@i; see the description of \plmac@newcommand@i on page \underline{17} to understand the ba-
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.

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

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

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\TeX macro, \perldo, instructs the Perl
sandbox to execute a block of code outside of all subroutines. \texttt{\perldo}'s implementation is much simpler than that of the other author macros because \texttt{\perldo} does not have to process subroutine arguments. Figure 3 illustrates the data that gets written to \texttt{\plmac@tofile} (indirectly) by \texttt{\perldo}.

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

\begin{verbatim}
\plmac@have@run@code
\plmac@run@code
\end{verbatim}

\begin{table}
\centering
\begin{tabular}{|c|}
\hline
RUN \\
\plmac@tag \\
\plmac@run@code \\
\plmac@ perlcode \\
\hline
\end{tabular}
\caption{Data written to \texttt{\plmac@tofile} to execute Perl code}
\end{table}

\texttt{\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 \texttt{\plmac@haveargs} but ends by invoking \texttt{\plmac@have@run@code} instead of \texttt{\plmac@havecode}.

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

\subsection{Communication between \LaTeX{} and Perl}

As shown in the previous section, when a document invokes \texttt{\perl[re]newcommand} to define a macro, \texttt{perltex.sty} defines the macro in terms of a call to \texttt{\plmac@write@perl}. In this section, we learn how \texttt{\plmac@write@perl} operates.
At the highest level, \LaTeX-to-Perl communication is performed via the filesystem. In essence, \LaTeX writes a file (\texttt{\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 .\texttt{tex} file (\texttt{\plmac@fromfile}); and, finally, \LaTeX reads and executes the new .\texttt{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—\texttt{\plmac@toflag}, \texttt{\plmac@fromflag}, and \texttt{\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, te\TeX, which is the \TeX distribution under which I developed Perl\TeX) cannot reliably poll for a file’s nonexistence; if a file is deleted in the middle of an \texttt{\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), Perl\TeX should behave as expected.

<table>
<thead>
<tr>
<th>Time</th>
<th>\LaTeX</th>
<th>Perl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write \texttt{\plmac@tofile}</td>
<td>Touch \texttt{\plmac@toflag}</td>
<td>Await \texttt{\plmac@toflag}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read \texttt{\plmac@tofile}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write \texttt{\plmac@fromfile}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delete \texttt{\plmac@toflag}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delete \texttt{\plmac@tofile}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delete \texttt{\plmac@doneflag}</td>
</tr>
<tr>
<td>Await \texttt{\plmac@fromflag}</td>
<td>Touch \texttt{\plmac@fromflag}</td>
<td>Await \texttt{\plmac@tofile}</td>
</tr>
<tr>
<td>Touch \texttt{\plmac@tofile}</td>
<td>Touch \texttt{\plmac@fromflag}</td>
<td>Delete \texttt{\plmac@fromflag}</td>
</tr>
<tr>
<td>Await \texttt{\plmac@doneflag}</td>
<td>Read \texttt{\plmac@fromfile}</td>
<td>Touch \texttt{\plmac@doneflag}</td>
</tr>
</tbody>
</table>

Figure 4: \LaTeX-to-Perl communication protocol

Although Figure 4 shows the read of \texttt{\plmac@fromfile} as the final step of the protocol, the file’s contents are in fact valid as soon as \LaTeX detects that \texttt{\plmac@fromflag} exists. Deferring the read to the end, however, enables Perl\TeX to support recursive macro invocations.

\begin{verbatim}
\plmac@await@existence
\if\plmac@file@exists
\plmac@file@existstrue
\else
\plmac@file@existsfalse
\fi
\end{verbatim}

The purpose of the \texttt{\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\epsilon’s \texttt{\IfFileExists} macro to check the file and invoke \texttt{\plmac@file@existstrue} or \texttt{\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.

\begin{verbatim}
\newif\ifplmac@file@exists
\newcommand{\plmac@await@existence}[]1]{%
  \begin{lrbox}{\@tempboxa}%
  \input\plmac@pipe
  \end{lrbox}%
  \loop
  \IfFileExists{#1}%
  {\plmac@file@existstrue}%
  {\plmac@file@existsfalse}%
  \ifplmac@file@exists
  \else
  \repeat
  }%\f้น
\plmac@outfile

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

\newwrite\plmac@outfile

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

\newcommand{\plmac@write@perl}(){%
  \begingroup
  \let\do@makeother\dospecials
  \catcode`\^^M=\active
  \newlinechar`\^^M
  \endlinechar`\^^M
  \catcode`\lbrace=1
  \catcode`\rbrace=2
  \plmac@write@perl@i
  \endgroup
}\end{verbatim}

\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.

\ifperl
\newcommand{\plmac@write@perl@i}{%

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

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

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

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

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

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

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

\plmac@await@existence\plmac@fromflag

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

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

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

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

26
\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 perltext.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 \fbox{Perl\TeX}. 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}.

\plmac@show@placeholder There’s really no point in outputting a framed “Perl\TeX” 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 “Perl\TeX” will be output.

\gdef\plmac@show@placeholder#1#2\@empty{%
  \ifx#1D\relax
    \endgroup
  \else
    \endgroup
    \fbox{Perl\TeX}\
  \fi
}%
\AtBeginDocument{%
  \renewcommand{\plmac@write@perl@i}{%
    \plmac@show@placeholder\empty%  
  }%
}%

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 perltext.sty, as detailed in Section 3.1), perltex.pl executes it within a secure sandbox and transmits the resulting \LaTeX code back to the document.

3.2.1 Header comments

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

#! /usr/bin/env perl

304 }
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.

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

3.2.3 Perl modules and pragmas

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

```perl
use Safe;
```
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 $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 $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 \LaTeX default of texput.
$toperl $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 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;
my $fromperl;
my $toflag;
my $fromflag;
my $doneflag;
my $logfile;
my $pipe;

Other global variables

@latexcmdline is the command line to pass to the \TeX\ executable. $styfile is the string noperltx.sty if perltex.pl is run with --makesty, otherwise undefined. @macroexpansions is a list of Perl\TeX\ macro expansions in the order they were encountered. It is used for creating a noperltx.sty file when --makesty is specified. $sandbox is a secure sandbox in which to run code that appeared in the \TeX\ document. $sandbox_eval is a subroutine that evaluates a string within $sandbox (normally) or outside of all sandboxes (if --nosafe is specified). $latexpid is the process ID of the \LaTeX\ process.

my @latexcmdline;
my $styfile;
my @macroexpansions;
my $sandbox = new Safe;
my $sandbox_eval;
my $latexpid;

$pipestring is a constant string to write to the $pipe named pipe (or file) at each \TeX\ synchronization point. Its particular definition is really a bug workaround for \Xe\TeX. The current version of \Xe\TeX\ 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 \texttt{\endinput} used in prior versions of Perl\TeX\ inserted a spurious \texttt{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.

my $pipestring = "\%\%\%\% Generated by \$progname\\\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 contents of the environment variable PERLTEX or the value “latex” if PERLTEX is not specified. We then use Getopt::Long to parse the command line, leaving any parameters we don’t recognize in the argument vector (@ARGV) because these are presumably latex options.

$latexprog = $ENV{"PERLTEX"} || "latex";
Getopt::Long::Configure("require_order", "pass_through");
GetOptions("help" => sub {pod2usage(-verbose => 1)},
"latex=s" => \\latexprog,
"safe!" => \\runsafely,
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.
"pipe!" => \\usepipe,
"synctext=s" => \\pipestring,
"makesty" => sub {
    \$styfile = "noperltex.sty"},
"permit=s" => @permittedops || pod2usage(2);

Preparing a \LaTeX{} command line

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

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

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.

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

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

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

\$latexcmdline\$firstcmd =
\$latexcmdline\$firstcmd;

### 3.2.6 Increasing Perl\TeX's robustness

\$toperl = File::Spec->rel2abs($toperl);
\$fromperl = File::Spec->rel2abs($fromperl);
\$toflag = File::Spec->rel2abs($toflag);
\$fromflag = File::Spec->rel2abs($fromflag);
\$logfile = File::Spec->rel2abs($logfile);
\$pipe = File::Spec->rel2abs($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.

\$SIG{"ALRM"} = sub {
  undef $latexpid;
  exit 0;
};

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).

\$SIG{"PIPE"} = "IGNORE";

#### delete_files

On some operating systems and some filesystems, deleting a file may not cause the file to disappear immediately. Because Perl\TeX\ 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.

sub delete_files (@) {
  foreach my $filename (@_) {
    unlink $filename;
    while (-e $filename) {
      unlink $filename;
      sleep 0;
    }
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.

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

### 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 `$styfile` is defined, a \LaTeX\ 2ε style file. As `$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.

```perl
delete_files($toperl, $fromperl, $toflag, $fromflag, $doneflag, $pipe);
open (LOGFILE, ">$logfile") || die "open("$logfile"): $!
";
autoflush LOGFILE 1;
if (defined $styfile) {
    open (STYFILE, ">$styfile") || die "open("$styfile"): $!
";
}
if (!$usepipe || !eval {mkfifo($pipe, 0600)}) {
    sysopen PIPE, $pipe, O_WRONLY|O_CREAT, 0755;
    autoflush PIPE 1;
    print PIPE $pipestring;
    close PIPE;
    $usepipe = 0;
}
defined ($latexpid = fork) || die "fork: $!
";
unshift @latexcmdline, $latexprog;
if (!$latexpid) {
    exec @{$latexcmdline}[0] @latexcmdline;
    die "exec('@latexcmdline'): $!
";
}
```
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.

```plaintext
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.

```plaintext
470 while (1) {
471   awaitexists($toflag);
472   my $entirefile;
473   {
474     local $/ = undef;
475     open (TOPERL, "<$toperl") || die "open($toperl): $!
476     $entirefile = <TOPERL>
477     close TOPERL;
478   }
479   $entirefile =~ s/\r//g;
480   my ($optag, $macroname, @otherstuff) =
481     map {chomp; $_} split "$separator\n", $entirefile;
```

```plaintext
$entirefile
```

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

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

```plaintext
$optag
$macroname
@otherstuff
```

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

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

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

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

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.

```perl
my $perlcode;
if ($optag eq "DEF") {
  $perlcode =
    sprintf "sub %s {%s}
", $macroname, @otherstuff[0];

\texttt{\L a T e X}: \verbatim
\pernewcommand{\mymacro}[2]{{
  sprintf "Isn't \$_[0] \%s \$_[1]\n", 
  \$_[0]>=\$_[1] ? ">
  : "<"
}}

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

Figure 5: Conversion from \LaTeX{} to Perl (subroutine definition)
\LaTeX: \texttt{\mymacro{12}{34}}

\begin{figure}[h]
\centering
\begin{verbatim}
\LaTeX: \\mymacro{12}{34}
\end{verbatim}
\vspace{1em}
\begin{verbatim}
Perl: \texttt{latex_mymacro ('12', '34');}
\end{verbatim}
\caption{Conversion from \LaTeX{} to Perl (subroutine invocation)}
\end{figure}

497 } 
498 elsif ($optag eq "USE") { 
499     $perlcode = sprintf "%s (%s);
499 \"macroname, join(\", \", @otherstuff); 
500 } 
501 elsif ($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{}'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 “\texttt{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 “\texttt{at (eval \langle number⟩) line \langle 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 \texttt{--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 \langle number\rangle line \langle number\rangle/; 
518         print LOGFILE "# ===> $warningmsg\n\n";
519     }
520 }
521 \$result = "" if !\$result || $optag eq "RUN";
if ($@) {
    my $msg = $@;
    $msg =~ s/at \(eval \d+\) line \d+\W+//;
    $msg =~ s/\n/\MessageBreak\n/g;
    $msg =~ s/\s+/ /;
    $result = "\\PackageError{perltex}{$msg}";
    my @helpstring;
    if ($msg =~ /\btrapped by\b/) {
        @helpstring =
            ("The preceding error message comes from Perl. Apparently,",
             "the Perl code you tried to execute attempted to perform an ",
             "'unsafe' operation. If you trust the Perl code (e.g., if",
             "you wrote it) then you can invoke perltex with the --nosafe",
             "option to allow arbitrary Perl code to execute.";
             "Alternatively, you can selectively enable Perl features",
             "using perltex's --permit option. Don't do this if you don't",
             "trust the Perl code, however; malicious Perl code can do a",
             "world of harm to your computer system.");
    }
    else {
        @helpstring =
            ("The preceding error message comes from Perl. Apparently,",
             "there's a bug in your Perl code. You'll need to sort that",
             "out in your document and re-run perltex.");
    }
    my $helpstring = join ("\MessageBreak\n", @helpstring);
    $helpstring =~ s/\./\space\space /g;
    $result .= \{$helpstring\};
} else {
    @helpstring =
        ("The preceding error message comes from Perl. Apparently,",
         "there's a bug in your Perl code. You'll need to sort that",
         "out in your document and re-run perltex.");
    my $helpstring = join ("\MessageBreak\n", @helpstring);
    $helpstring =~ s/\./\space\space /g;
    $result .= \{$helpstring\};
}
if (open (PIPE, ">$pipe")) {
    autoflush PIPE 1;
    print PIPE $pipestring;
    close PIPE;
}

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

```
awaitexists($toperl);
delete_files($fromflag);
open (DONEFLAG, ">$doneflag") || die "open($doneflag): $!\n";
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).

```
alarm 1;
if (open (PIPE, ">$pipe")) {
    autoflush PIPE 1;
    print PIPE $pipestring;
    close PIPE;
}
alarm 0;
```

### 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.

```
END {
    close LOGFILE;
    if (defined $latexpid) {
        kill (9, $latexpid);
        exit 1;
    }
    if (defined $styfile) {
        print STYFILE <<"STYFILEHEADER1";
        \NeedsTeXFormat{LaTeX2e}[1999/12/01]
        \ProvidesPackage{noperltex}[2007/09/29 v1.4 Perl-free version of PerlTeX specific to $jobname.tex]
        STYFILEHEADER1
    }
    ;
    print STYFILE <<'STYFILEHEADER2';
```

38
\RequirePackage{filecontents}
\let\noperltex@PackageError=\PackageError
\renewcommand{\PackageError}[3]{\relax}
\RequirePackage{perltex}
\let\PackageError=\noperltex@PackageError
\plmac@macro@invocation@num
\plmac@show@placeholder
noperltex.sty works by redefining the \plmac@show@placeholder macro, which normally outputs a framed “Perl\TeX” 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 Perl\TeX-defined macro.

% Modify \plmac@show@placeholder to input the next noperltex-*.tex file
% each time a Perl\TeX-defined macro is invoked.
\newcount\plmac@macro@invocation@num
\gdef\plmac@show@placeholder#1#2\@empty{%
  \ifx#1U\relax
    \endgroup
    \advance\plmac@macro@invocation@num by 1\relax
    \global\plmac@macro@invocation@num=\plmac@macro@invocation@num
    \input{noperltex-\the\plmac@macro@invocation@num.tex}%
  \else
  \fi
}
\@empty}

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.

foreach my $e (0 .. $#macroexpansions) {
  print STYFILE "\n";
  printf STYFILE "\% Invocation #\d\n", 1+$e;
  printf STYFILE "\begin{filecontents}{noperltex-%d.tex}\n", 1+$e;
  print STYFILE $macroexpansions[$e], "\endinput\n";
  print STYFILE "\end{filecontents}\n";
}

print STYFILE "\endinput\n";
close STYFILE;

exit 0;
}

__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 \texttt{--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 \texttt{⟨some-language-other-than-Perl⟩} \TeX{}. Fortunately, porting Perl\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\TeX”) as per the Perl\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 \texttt{⟨some-language-other-than-Perl⟩}\TeX{} alongside Perl\TeX{}, 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 \url{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 XeT\TeX\ 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

v1.0
  General: Initial version 1
v1.0a
  General: Made all \texttt{unlink} calls wait for the file to actually disappear 27
  Undefined $/ only locally 34
  \texttt{waitexists}: Bug fix: Added \texttt{"undef \$latexpid"} to make the END block correctly return a status code of 0 on success 33
v1.1
  General: Added new \texttt{perlnewenvironment} and \texttt{perlrenewenvironment} macros 21
  \texttt{\plmac@havecode}: Added a \texttt{\plmac@next} hook to support PerlTEX’s new environment-defining macros 19
  \texttt{\plmac@write@perl@i}: Added a dummy version of the macro to use if \texttt{latex} was launched directly, without \texttt{perltx.pl} 27
  Made argument-handling more rational by making \texttt{\protect}, \texttt{\begin}, and \texttt{\end} non-expandable 26
v1.2
  General: Renamed \texttt{perlmicros.sty} to \texttt{perltx.sty} for consistency 1
  \texttt{\plmac@write@perl@i}: Moved the \texttt{\input} of the generated Perl code to the end of the routine in order to support recursive PerlTEX macro invocations 26
v1.3
  General: Modified \texttt{perltx.pl} to eschew the sandbox altogether when --\texttt{nosafe} is specified 28
  \texttt{\perldo}: Introduced \texttt{\perldo} to support code execution outside of all subroutines 23
  \texttt{\plmac@run@code}: Added to assist \texttt{\perldo} 23
v1.4
  General: Added support for a --\texttt{makesty} option that generates a PerlTEX-free style file called \texttt{noperltx.sty} 38
v1.5
  \texttt{\plmac@file@existsfalse}: Modified to read from a named pipe before checking file existence 30
v1.6
  General: Added an undocumentated --\texttt{nopipe} option to \texttt{perltx.pl} to help it work with XeT\TeX\ 30
v1.7
  General: Added an undocumentated --\texttt{synctext} option to alter the text written to \texttt{\$pipe} 30
  \texttt{\$pipestring}: Introduced this variable as a workaround for XeT\TeX\’s attempt to rewind \texttt{\$pipe} 30
v1.8
\plmac@requiredfalse: Introduced an optional package option to suppress the “must be compiled using perltex” error message. 13
\plmac@write@perl@i: Renamed \ifplmac@have@perltex to \ifperl to help authors write mixed \LaTeX/Perl\TeX documents. 26

v1.9
General: Introduced handlers for \texttt{signalrm} and \texttt{sigpipe} to make \texttt{perltex.pl} more robust to \texttt{latex} exiting at an inopportune time. 32
\texttt{delete_files}: Replaced all \texttt{unlink...while -e} statements with calls to a new \texttt{delete_files} subroutine. 32
\plmac@await@existence: Put the \texttt{\input\plmac@pipe} within an \texttt{lrbox} environment to prevent a partial read from introducing spurious text into the document. 24
\texttt{awaitexists}: Hoisted $awaitexists from the main loop and made it a top-level subroutine. 33

v2.0
General: Refer to each communication file using its absolute path. This makes \texttt{perltex.pl} robust to user code that changes the current directory. 32
$\texttt{msg}$: Substituted \texttt{\MessageBreak} for newline when reporting error messages produced by user code. 36

v2.1
General: Replaced \texttt{abs\path()} with \texttt{File::Spec->rel2abs()} because the latter seems to be more robust to nonexistent files. 32
\texttt{@otherstuff}: Normalized line endings across Unix/Windows/Macintosh. 34

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

\$ ........................................ 370–373 \texttt{closeout} .................. 276, 279, 282
\% .................................. 11X, 54X, 366
& ........................................ 468
\@tempboxa .................................. 247 \texttt{DeclareOption} ............. 65
\{ .................................. 20X, 134, 228, 265 \texttt{delete_files} ............. 421
\} .................................. 20X, 135, 229, 266 \texttt{do} .......................... 130, 224, 261
^ .... 131–133, 139, 225–227, 262–264 \texttt{dolinkwords} .......... 8X, 42X
$\texttt{doneflag}$ .......................... 351 \texttt{\dospecials} .......... 130, 224, 261
\texttt{\afterassignment} .......................... 136, 230
\texttt{\AtBeginDocument} ................ 296
\texttt{awaitexists} .......................... 431
\texttt{\endinput} .......................... 554
\texttt{\endlinechar} .......................... 133, 227, 264
$\texttt{\entirefile}$ .......................... 471
\texttt{\BODY} .......................... 42X
$\texttt{\environment\ (package)}$ ............. 10
\plmac@pipe .................. 248, 407
\plmac@requiredfalse ........ 64
\plmac@requiredtrue ........ 64
\plmac@run@code .............. 233
\plmac@sep .................. 139
                      144–147, 153, 154, 163, 236–239
\plmac@show@placeholder 288, 298, 601
\plmac@starchar ...... 102, 171, 178, 198
\plmac@tag .................. 145, 147, 154, 163, 237, 239, 401
\plmac@tofile .............. 271, 281, 402
\plmac@toflag .............. 278, 404
\plmac@write@perl ......... 144, 173, 180, 236, 259
\plmac@write@perl@i ...... 267, 269, 286

$proname .................. 351
$runsafely .................. 347
$sandbox .................. 360
$sandbox_eval .............. 360
$separator ................. 388
SIGALRM .................. 32
SIGPIPE .................. 32
\stopwords .................. 52X, 54X
$styfile .................. 360

$toflag .................. 351
$toperl .................. 351
\typeout .................. 87, 88

\usepackage ................. 2–4X
$usepipe .................. 347

\write .................. 275