Perl\TeX: Defining \LaTeX macros in terms of Perl code*

Scott Pakin
scott+pt@pakin.org
September 14, 2019

Abstract

Perl\TeX 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. Perl\TeX 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.

Perl\TeX'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. Perl\TeX 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 Perl\TeX:

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

*This document corresponds to Perl\TeX v2.2, dated 2019/09/14.
Then, executing \texttt{\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\TeX makes it easy to export this to \LaTeX:

\begin{verbatim}
\perlnewcommand{\substr}[3]{substr \$_[0], \$_[1], \$_[2]}
\end{verbatim}

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

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

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:

\begin{verbatim}
\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 '%d', \$i+\$j+1) : '1';
    }
    \$result .= join (' & ', @row) . ' \\
  }\$result .= ' end{array}
\end{array}';
  return \$result;
}
\hilbertmatrix{20}
\end{verbatim}

\begin{verbatim}
2
\end{verbatim}
example would have been much more difficult to implement without PerlTEX:

\begin{tabular}{@{}r|*{10}{r}@{}}
\rownum & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\rownum & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\rownum & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{tabular}

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

\begin{verbatim}
\ perlnewcommand{\texttt{\textbackslash newcounter}{ssrow}}
\ perlnewenvironment{\texttt{\textbackslash 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\}r\{c\}r\{c\}r\{c\}r\{c\}r\{c\}r\{c\}r\{c\}r\{c\}r\{c\}r\{c\}\n";
$tabular .= "\multicolumn\{1\}\{c\}\{c\} & \"; "$\n";

foreach (1 .. $cols) {
    $tabular .= "\multicolumn\{1\}\{c\}\{c\}c";
    $tabular .= "\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\\n";
    $tabular .= "r\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\\n";
    $tabular .= "r\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\\n";
    $tabular .= "r\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\\n";
    $tabular .= "r\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\}\{c\\n";
    $tabular .= "r\{c\}\{c\}\{c\}\{c\}\{c\\n";
    $tabular .= "r\{c\}\{c\}\{c\}\{c\\n"
}
else {
    $tabular .= "&\"; 
}
$tabular .= "\n";
\end{verbatim}
2 Usage

There are two components to using Perl\TeX. 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

perltex.sty defines five macros: \perlnewcommand, \perlrenewcommand, \ perlnewenvironment, \ perlrenewenvironment, and \perldo. The first four of these behave exactly like their \LaTeX\ 2\epsilon 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 Perl\TeX-defined macro or environments is converted to a Perl subroutine named after the macro/environment but beginning with "latex_". For example, a Perl\TeX-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, Perl\LaTeX 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 \ref{sec:sandbox}.) Having a secure sandbox implies that it is safe to build Perl\LaTeX 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:

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

\begin{verbatim*}
x was set to 123. x was set to 456. x was set to 789.
\end{verbatim*}

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

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

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

2.2 Making \texttt{perltx.pl} optional

Normally, \texttt{perltx.sty} issues a Document must be compiled using perltx error if a document specifies \texttt{\usepackage{perltx}} but is not compiled using \texttt{perltx.pl}. However, sometimes Perl\LaTeX may be needed merely to enhance a
optional Document was compiled without using the perltex script without aborting 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 Perl\TeX{} 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


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 \perlrnewcommand 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 \perlrnewcommand 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 \texttt{\usepackage{perltex}} line with \texttt{\usepackage{noperltex}} produces the same output but does not require \TeX{}, making the document suitable for distribution to people who do not have \TeX{} installed. The disadvantage is that \texttt{noperltex.sty} is specific to the document that produced it. Any changes to the document’s \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 \texttt{latex} (or whatever program was specified with --latex), including, for instance, the name of the \texttt{.tex} file to compile.

EXAMPLES

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

```
perltex myfile.tex
```

To use \texttt{pdflatex} instead of regular \texttt{latex}, use the -latex option:

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

If \LaTeX{} gives a “trapped by operation mask” error and you trust the \texttt{.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\LaTeX{} 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\LaTeX{}.

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\LaTeX{} macro \texttt{\textbackslash dolinkwords}, which takes three arguments: a URL template that contains \texttt{\%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 \texttt{\langle letters \rangle}, \texttt{\begin{(letters)}}, or \texttt{\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{\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...\else...\fi} is used to surround the definition of the \texttt{\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 \texttt{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 \texttt{\usepackage[optional]{perltex}} line with
the following \LaTeX code:

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

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 PerlT\EX 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 PerlT\EX’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   %subs = ();
19X   $replace = sub \{subs{$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]+)/i, \$text) {
26X     $lword = lc $word;
27X     if (defined $stopwords{$lword} || $lword =~ /[^a-z]/) {
28X       $newtext .= $word;
29X     } else {
30X       $newtext .= sprintf "\href{$url}{%s}\"", $lword, $word;
31X     }
32X   }
33X}
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 303 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 perltext.sty

Although I’ve written a number of LATEX packages, `perltext.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 \relaxed variable be “#” right before invoking the macro. This technique appears in \plmac@havecode.

I hope you find reading the `perltext.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 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, `perltext.sty` issues an error message if the document is compiled without using perltext.pl. When optional is specified, `perltext.sty` suppresses the error message.

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

\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. `perltext.pl`
therefore defines \texttt{plmac@tag} as a long string of random uppercase letters. \texttt{plmac@tofile} is the name of a file used for communication from \LaTeX{} to Perl. \texttt{plmac@fromfile} is the name of a file used for communication from Perl to \LaTeX{}. \texttt{plmac@toflag} signals that \texttt{plmac@tofile} can be read safely. \texttt{plmac@fromflag} signals that \texttt{plmac@fromfile} can be read safely. \texttt{plmac@doneflag} signals that \texttt{plmac@fromflag} has been deleted. Table 1 lists all of these variables along with the value assigned to each by \texttt{perltex.pl}.

Table 1: Variables used for communication between Perl and \LaTeX{}

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

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

\begin{verbatim}
\ifperl
\perltrue
\def\plmac@tag{}{if\texttt{plmac@tag} undefined\texttt{let}\plmac@tag=\texttt{relax}}{}
\else
\ifplmac@required
\PackageError{perltex}{Document must be compiled using perltex}
\{Instead of compiling your document directly with latex, you need to use the perltex script. perltex sets up a variety of macros needed by\MessageBreak to \LaTeX{} and Perl.\}
\else
\bgroup
\obeyspaces
\typeout{perltex: Document was compiled without using the perltex script;}
\typeout{ it may not print as desired.}
\egroup
\end{verbatim}
3.1.2 Defining Perl macros

PerL\TeX\ defines five macros intended to be called by the author. Section 3.1.2 details the implementation of two of them: \perlnewcommand and \perlrernewcommand. (Section 3.1.3 details the implementation of the next two, \perlnenviron and \perlrerenviron; 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 writeable version of the name in \plmac@cleaned@macname, and the macro's previous definition (needed by \perlrenewcommand) in \plmac@oldbody. Finally, \plmac@newcommand@i invokes \plmac@newcommand@ii.

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

4. \plmac@newcommand@i@ii@opt defines \plmac@defarg as the default value of the optional argument. \plmac@newcommand@i@ii@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
Figure 1: Data written to \plmac@tofile to define a Perl subroutine

```
USE
\plmac@tag
\plmac@cleaned@macname
\plmac@tag
#1
\plmac@tag
#2
\plmac@tag
#3
...
#⟨last⟩
```

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

the user’s \LaTeX macro causes \plmac@write@perl to pass the information shown in Figure 2 to perltex.pl.

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.

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Sample definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\plmac@command</code></td>
<td><code>\newcommand</code></td>
</tr>
<tr>
<td><code>\plmac@starchar</code></td>
<td><code>*</code></td>
</tr>
<tr>
<td><code>\plmac@macname</code></td>
<td><code>\example</code></td>
</tr>
<tr>
<td><code>\plmac@cleaned@macname</code></td>
<td><code>\example</code></td>
</tr>
<tr>
<td></td>
<td>(catcode 11)</td>
</tr>
<tr>
<td><code>\plmac@oldbody</code></td>
<td><code>\relax</code></td>
</tr>
<tr>
<td></td>
<td>(presumably)</td>
</tr>
<tr>
<td><code>\plmac@numargs</code></td>
<td>3</td>
</tr>
<tr>
<td><code>\plmac@defarg</code></td>
<td><code>frobozz</code></td>
</tr>
<tr>
<td><code>\plmac@perlcode</code></td>
<td><code>\texttt{join(&quot;---&quot;, \_ \_ \_)}</code> (catcode 11)</td>
</tr>
</tbody>
</table>

If the user invoked `\perlnewcommand*` then `\plmac@newcommand@i` is passed a “*” and, in turn, defines `\plmac@starchar` as “*”. If the user invoked `\perlnewcommand` (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 `\perlrerenumcommand`.) It then invokes `\plmac@newcommand@ii`.

If the user invoked `\perlrenewcommand*` then `\plmac@newcommand@i` is passed a “*” and, in turn, defines `\plmac@starchar` as “*”. If the user invoked `\perlrenewcommand` (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 `\perlrerenumcommand`.) It then invokes `\plmac@newcommand@ii`.

If the user invoked `\perlnewcommand*` then `\plmac@newcommand@i` is passed a “*” and, in turn, defines `\plmac@starchar` as “*”. If the user invoked `\perlnewcommand` (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 `\perlrerenumcommand`.) It then invokes `\plmac@newcommand@ii`.
argument is to be optional or \plmac@newcommand@iii@no@opt if all arguments are to be mandatory.

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

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
\plmac@perlcode
\endgroup
}
Control is transferred to \texttt{plmac@havecode} from \texttt{plmac@haveargs} right after the user’s macro body is assigned to \texttt{plmac@perlcode}. We now have everything we need to define the user’s macro. The goal is to define it as “\texttt{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 \texttt{plmac@perlcode} piece-by-piece.

\textbf{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 \texttt{plmac@sep} as a carriage-return character of category code 11 (“letter”).

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

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

\texttt{\newcount plmac@argnum}

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

\texttt{\def plmac@havecode{\% \endgroup \plmac@define@sub}}

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

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

\textbf{plmac@body} The rest of \texttt{plmac@havecode} is preparation for defining the user’s macro. (\LaTeX{}’s \texttt{\newcommand} or \texttt{\renewcommand} will do the actual work, though.) \texttt{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).

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

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

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

\plmac@define@command

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

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

20
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.

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 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: \langle name \rangle and \end{name}. Because we want to reuse as much code as possible the idea is to define the “begin” code as one macro, then inject—by way of \plmac@next—a call to \plmac@end@environment, which defines the “end” code as a second macro.

\def\perlnewenvironment{}%\let\plmac@command=\newcommand\let\plmac@next=\plmac@end@environment%\@ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
The \plmac@newenvironment@i macro is analogous to \plmac@newcommand@i; see the description of \plmac@newcommand@i on page \pageref{newcommand} to understand the basic structure. The primary difference is that the environment name (#2) is just text, not a control sequence. We store this text in \plmac@envname to facilitate generating the names of the two macros that constitute an environment definition. Note that there is no \plmac@newenvironment@ii; control passes instead to \plmac@newcommand@ii.

Recall that an environment definition is a shortcut for two macro definitions: \texttt{\langle name\rangle} and \texttt{\end{\langle name\rangle}} (where \texttt{\langle name\rangle} was stored in \plmac@envname by \plmac@newenvironment@i). After defining \texttt{\langle name\rangle}, \plmac@havecode transfers control to \plmac@end@environment because \plmac@next was let-bound to \perl[re]newenvironment. \plmac@end@environment’s purpose is to define \texttt{\end{\langle name\rangle}}. This is a little tricky, however, because \LaTeX’s \texttt{\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 \texttt{\end{\langle name\rangle}} to it. Other than that, \plmac@end@environment is a combined and simplified version of \perlnewenvironment, \perlrenewenvironment, and \plmac@newenvironment@i.

\def\plmac@end@environment{%
\expandafter\def\expandafter\plmac@next\expandafter{%
\let\csname end\plmac@envname\endcsname=\plmac@end@macro
\let\plmac@next=\relax
}%
\def\plmac@macname{\plmac@end@macro}%
\expandafter\let\expandafter\plmac@oldbody\csname end\plmac@envname\endcsname
\expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
\expandafter\string\plmac@macname

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

\texttt{perldo} \hspace{1cm} Execute a block of Perl code and pass the result to \texttt{\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@have@code}.

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

\texttt{plmac@have@run@code} \hspace{1cm} Pass a block of code to Perl to execute. \texttt{plmac@have@run@code} is identical to \texttt{plmac@have@code} but specifies the \texttt{RUN} tag instead of the \texttt{DEF} tag.

\begin{verbatim}
def plmac@have@run@code{\% 
  \endgroup 
  \edefplmac@run@code{\% 
  \noexpandplmac@write@perl\{RUN\plmac@sep 
  \plmac@tag\plmac@sep 
  N/A\plmac@sep 
  \plmac@tag\plmac@sep 
  \the\plmac@perlcode 
  }\% 
  }\% 
\end{verbatim}
3.1.5 Communication between \LaTeX\ and Perl

As shown in the previous section, when a document invokes \texttt{\verb|\perl|}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 .tex file (\texttt{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—\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, \texttt{etex}, which is the \TeX\ distribution under which I developed \texttt{Perl\TeX}) cannot reliably poll for a file’s nonexistence; if a file is deleted in the middle of an \texttt{\immediate\openin}, \texttt{latex} aborts with an error message. These restrictions led to the regretfully 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), \texttt{Perl\TeX} should behave as expected.

\begin{figure}[h]
\centering
\begin{tabular}{|c|}
\hline
Time & \LaTeX\ & Perl \\
\hline
Write \texttt{plmac@tofile} & \texttt{Await plmac@toflag} & \texttt{Await plmac@toflag} \\
Touch \texttt{plmac@toflag} & \texttt{Read plmac@tofile} & \texttt{Read plmac@tofile} \\
Touch \texttt{plmac@tofile} & \texttt{Write plmac@fromfile} & \texttt{Write plmac@fromfile} \\
Await \texttt{plmac@fromflag} & \texttt{Delete plmac@toflag} & \texttt{Delete plmac@toflag} \\
\hline
\end{tabular}
\caption{\LaTeX\-to-Perl communication protocol}
\end{figure}

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...
The purpose of the \plmac@await@existence macro is to repeatedly check
the existence of a given file until the file actually exists. For conve-
nience, we use \LaTeXe’s \texttt{IfFileExists} macro to check the file and invoke
\plmac@file@existstrue or \plmac@file@existsfalse, as appropriate.

As a performance optimization we \texttt{\input} a named pipe. This causes the
\texttt{latex} process to relinquish the CPU until the \texttt{perltex} process writes data (always
just a comment plus \texttt{\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 \texttt{\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 \LaTeXe and Perl.
When \texttt{\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 \texttt{\plmac@write@perl@i} to send its argument to Perl and read back the (\LaTeX) result.

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

\begin{verbatim}
\ifperl
  \newcommand{\plmac@write@perl@i}[1]{%
    \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

    \begin{protocol}
      \begin{verbatim}
      \textbf{The first step is to write argument \#1 to \plmac@tofile:}
      \begin{verbatim}
      \immediate\openout\plmac@toflag=\plmac@toflag\relax
      \immediate\closeout\plmac@toflag

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

      \begin{verbatim}
      \immediate\openout\plmac@tofile=\plmac@toflag\relax
      \immediate\closeout\plmac@tofile
      \end{verbatim}

      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 \texttt{PerlT\TeX} macros.

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

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 “\LaTeX”. 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}`.

\newcommand{\plmac@write@perl@i}[1]{\endgroup}
\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}[1]{%
    \plmac@show@placeholder#1\@empty
    }%
}\fi

### 3.2 `perltex.pl`

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

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

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

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

# This is file `perltex.pl',
# generated with the docstrip utility.
#
# perltex.dtx (with options: `perltex')
#
# This is a generated file.
#
# Copyright (C) 2003-2019 Scott Pakin <scott+pt@pakin.org>
#
# This file may be distributed and/or modified under the conditions
# of the LaTeX Project Public License, either version 1.3c of this
# license or (at your option) any later version. The latest
# version of this license is in:
#
# http://www.latex-project.org/lppl.txt
#
# and version 1.3c or later is part of all distributions of LaTeX
# version 2006/05/20 or later.
```

3.2.2 Top-level code evaluation

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

```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;
use Opcode;
use Getopt::Long;
use Pod::Usage;
use File::Basename;
use Fcntl;
use POSIX;
use File::Spec;
use IO::Handle;
use warnings;
use strict;
```

3.2.4 Variable declarations

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

Variables corresponding to command-line arguments

- `$latexprog` is the name of the \LaTeX\ executable (e.g., “latex”). If `$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.

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

Filename variables

- `$proname` is the run-time name of the perltex.pl program. `$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

```perl
my $proname = ...
my $jobname = ...
my $toperl = ...
my $fromperl = ...
my $toflag = ...
my $fromflag = ...
my $doneflag = ...
my $logfile = ...
my $pipe = ...
```
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.

353 my $progname = basename $0;
354 my $jobname = "texput";
355 my $toperl;
356 my $fromperl;
357 my $toflag;
358 my $fromflag;
359 my $doneflag;
360 my $logfile;
361 my $pipe;

Other global variables

@latexcmdline
$styfile
@macroexpansions
$sandbox
$sandbox_eval
$latexpid
$pipestring

@latexcmdline is the command line to pass to the \LaTeX 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 \LaTeX 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.

362 my @latexcmdline;
363 my $styfile;
364 my @macroexpansions;
365 my $sandbox = new Safe;
366 my $sandbox_eval;
367 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 Xe\LaTeX. The current version of Xe\LaTeX 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 Perl\TeX 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.

368 my $pipestring = "\\\\\\\\\\\\\\\\\\\\\\\\\\\\\ Generated by $progname\\\\\\\\\\\\\\\\\\\\\\\\\\\\"

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 perltext.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 {\$stylefile = "noperltex.sty"},
"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 \"\". This string, which represents a filename, is used to set $jobname.

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

```
$separator To avoid conflicts with the code and parameters passed to Perl from \LaTeX{} (see Figure 1 on page 10 and Figure 2 on page 10) 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";
```
395 $fromperl = $jobname . ".frpl";
396 $toflag = $jobname . ".tfpl";
397 $fromflag = $jobname . ".ffpl";
398 $doneflag = $jobname . ".dfpl";
399 $logfile = $jobname . ".lgpl";
400 $pipe = $jobname . ".pipe";

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

401 $latexcmdline[$firstcmd] =
402 sprintf '\makeatletter' . '\def%s{%s}' x 7 . '\makeatother%s',
403 '\plmac@tag', $separator,
404 '\plmac@tofile', $toperl,
405 '\plmac@fromfile', $fromperl,
406 '\plmac@toflag', $toflag,
407 '\plmac@fromflag', $fromflag,
408 '\plmac@doneflag', $doneflag,
409 '\plmac@pipe', $pipe,
410 $latexcmdline[$firstcmd];

3.2.6 Increasing PerlT\TeX’s robustness

411 $toperl = File::Spec->rel2abs($toperl);
412 $fromperl = File::Spec->rel2abs($fromperl);
413 $toflag = File::Spec->rel2abs($toflag);
414 $fromflag = File::Spec->rel2abs($fromflag);
415 $doneflag = File::Spec->rel2abs($doneflag);
416 $logfile = File::Spec->rel2abs($logfile);
417 $pipe = File::Spec->rel2abs($pipe);

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

418 $SIG{"ALRM"} = sub {
419     undef $latexpid;
420     exit 0;
421 };

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).

422 $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.

423 sub delete_files (@)
424 {  
425     foreach my $filename (@_) {  
426         unlink $filename;  
427         while (-e $filename) {  
428             unlink $filename;  
429             sleep 0;  
430         }  
431     }  
432 }

awaitexists  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.

433 sub awaitexists ($) {  
434     while (!-e $_[0]) {  
435         sleep 0;  
436         if (waitpid($latexpid, &WNOHANG)==-1) {  
437             delete_files($toperl, $fromperl, $toflag,  
438             $fromflag, $doneflag, $pipe);  
439             undef $latexpid;  
440             exit 0;  
441         }  
442     }  
443 }  
444

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 \texttt{\textbackslash endinput} if we can't create a named pipe—and, if $styfile is defined, a \LaTeX\ 2\epsilon style file. As @latexcmdline contains the complete command line to pass to \texttt{latex} we need only fork a new process and have the child process overlay itself with \texttt{latex}. perltex.pl continues running as the parent.

445 delete_files($toperl, $fromperl, $toflag, $fromflag, $doneflag, $pipe);  
446 open (LOGFILE, ">$logfile") || die "open("$logfile"): !\n";  
447 autoflush LOGFILE 1;  
448 if (defined $styfile) {  
449     open (STYFILE, ">$styfile") || die "open("$styfile"): !\n";  
450 }

451 if (!$usepipe || !eval {mkfifo($pipe, 0600)}) {  
452     sysopen PIPE, $pipe, O_WRONLY|O_CREAT, 0755;  
453     autoflush PIPE 1;  
454     print PIPE $pipestring;  
455     close PIPE;  
456     $usepipe = 0;  
457 }

458 defined ($latexpid = fork) || die "fork: !\n";
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.

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.
then \texttt{@otherstuff} will be a list of subroutine arguments. If \texttt{$optag}$ is \texttt{RUN} then \texttt{@otherstuff} will be a block of Perl code to run.

481 $\texttt{entirefile} = s/\r//g;
482 my ($\texttt{optag}$, $\texttt{macroname}$, \texttt{@otherstuff}) =
483 map {chomp; $_} split "$\texttt{separator}\n", $\texttt{entirefile};

We clean up the macro name by deleting all leading non-letters, replacing all subsequent non-alphanumerics with \texttt{"_"}, and prepending \texttt{"latex_"} to the macro name.

484 $\texttt{macroname} =~ s/^[^A-Za-z]+//;
485 $\texttt{macroname} =~ s/\W/_/g;
486 $\texttt{macroname} = "latex_" . $\texttt{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 \texttt{“\"} with \texttt{"\\"} and every occurrence of \texttt{“\’"} with \texttt{"\’"}.

487 if ($\texttt{optag}$ eq "USE") {
488   foreach (\texttt{@otherstuff}) {
489     s/\"/\\/g;
490     s/\'/\’/g;
491     $\_ = "'\$\_';"
492   }
493 }

\texttt{$perlcode}$ There are three possible values that can be assigned to \texttt{$perlcode$}. If \texttt{$optag$} is \texttt{DEF}, then \texttt{$perlcode$} is made to contain a definition of the user’s subroutine, named \texttt{$macroname$}. If \texttt{$optag$} is \texttt{USE}, then \texttt{$perlcode$} becomes an invocation of \texttt{$macroname$} which gets passed all of the macro arguments. Finally, if \texttt{$optag$} is \texttt{RUN}, then \texttt{$perlcode$} is the unmodified Perl code passed to us from \texttt{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.

494 my \texttt{$perlcode$};
495 if ($\texttt{optag}$ eq "DEF") {
496   \texttt{$perlcode$} =
497   sprintf "sub \%s {%s}\n", $\texttt{macroname}, \texttt{$otherstuff}[0];
498 }
499 elsif ($\texttt{optag}$ eq "USE") {
500   \texttt{$perlcode$} = sprintf "%s (%s);\n", $\texttt{macroname}, join("", \texttt{@otherstuff});
501 }
502 elsif ($\texttt{optag}$ eq "RUN") {
503   \texttt{$perlcode$} = $\texttt{otherstuff}[0];
504 }
505 } else {
506   die "$\texttt{progname}$: Internal error -- unexpected operation tag \"$\texttt{optag}\"\n";
507 }

35
Log what we’re about to evaluate.

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

We’re now ready to execute the user’s code using the `$sandbox_eval` function.

```perl
511  undef $_;
512  my $result;
513  {
514      my $warningmsg;
515      local $SIG{__WARN__} = sub {chomp ($warningmsg=$_[0]); return 0};
516      $result = $sandbox_eval->($perlcode);
517      if (defined $warningmsg) {
```

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{} 2ε’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 \texttt{--makesty}).
519  $warningmsg =~ s/at \(eval \d+\) line \d+\W+//;
520  print LOGFILE "# ==> $warningmsg\n\n";
521  }
522  }
523  $result = "" if !$result || $optag eq "RUN";
524  if ($@) {
525      my $msg = $@;
526      $msg =~ s/at \(eval \d+\) line \d+\W+//;
527      $msg =~ s/\n/MessageBreak\n/g;
528      $msg =~ s/\s+/ /;
529      $result = "\PackageError{perltex}{$msg}";
530      my @helpstring;
531      if ($msg =~ /trapped by\b/) {
532          @helpstring =
533          ("The preceding error message comes from Perl. Apparently," ,
534           "the Perl code you tried to execute attempted to perform an",
535           "'unsafe' operation. If you trust the Perl code (e.g., if ",
536           "you wrote it) then you can invoke perltex with the --
537           nosafe",
538           "option to allow arbitrary Perl code to execute. ",
539           "Alternatively, you can selectively enable Perl features",
540           "using perltex's --permit option. Don't do this if you don't",
541           "trust the Perl code, however; malicious Perl code can do a",
542           "world of harm to your computer system.");
543      }
544      else {
545          @helpstring =
546          ("The preceding error message comes from Perl. Apparently," ,
547           "there's a bug in your Perl code. You'll need to sort that",
548           "out in your document and re-run perltex.");
549      }
550      my $helpstring = join ("MessageBreak\n", @helpstring);
551      $helpstring =~ s/\. /.\space\space /g;
552      $result .= "\{$helpstring}\";
553  }
554  push @macroexpansions, $result if defined $styfile && $optag eq "USE";

Log the resulting \LaTeX{} code.

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

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

$\texttt{result} := '\\endinput';

Continuing the protocol described in Figure 4 on page 24 we now write $\texttt{result}$ (which contains either the result of executing the user's or a \texttt{PackageError}) to the $\texttt{fromperl}$ file, delete $\texttt{toflag}$, $\texttt{toperl}$, and $\texttt{doneflag}$, and notify \LaTeX{} by touching the $\texttt{fromflag}$ file. As a performance optimization, we also write \texttt{\endinput} into $\texttt{pipe}$ to wake up the \texttt{latex} process.
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.

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 \texttt{SIGALRM} handler (page \pageref{lint:exception-signal}).

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.

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

% Suppress the "Document must be compiled using perltex" error from perltex.
\let
operltex@PackageError=\PackageError
\renewcommand{\PackageError}[3]{}
\RequirePackage{perltex}
\let\PackageError=\noperltex@PackageError
\let\plmac@macro@invocation@num
\plmac@show@placeholder
noperltex.sty works by redefining the \plmac@show@placeholder macro, which
normally outputs a framed “PerlT EX” 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 PerlT EX-defined macro.

% Modify \plmac@show@placeholder to input the next noperltex-*.tex file
% each time a PerlTeX-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
\endgroup
\fi
\}
STYFILEHEADER2
;

\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";
}
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 \langle some-language-other-than-Perl\rangle \LaTeX{}. Fortunately, porting Perl\LaTeX{} 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\LaTeX”) as per the Perl\LaTeX{} 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 \langle some-language-other-than-Perl\rangle \LaTeX{} alongside Perl\LaTeX{}, enabling multiple programming languages to be utilized in the same \LaTeX{} document.

4 License agreement

Copyright © 2003–2019 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 Perl\TeX\-free \texttt{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 Perl\TeX\ with \Xe\TeX\ and to Jonathan Kew for suggesting a workaround; to Linus Källberg for reporting and helping diagnose some problems with running Perl\TeX\ on Windows; and to Ulrike Fischer for reporting and helping correct a bug encountered when using \texttt{noperltex.sty} with newer versions of \LaTeX\. Also, thanks to the many people who have sent me fan mail or submitted bug reports, documentation corrections, or feature requests. (The \texttt{\perldo} macro and the \texttt{--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{awaitexists}: Bug fix: Added
"\texttt{undef \latexpid}"
to make the
\texttt{END} block correctly return
a status code of 0 on success 33

v1.1
\texttt{\plmac@havecode}: Added a
\texttt{\plmac@next} hook to support
Perl\TeX\’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{perltex.pl} 27
Made argument-handling more
rational by making \texttt{\protect},
\texttt{\begin}, and \texttt{\end}
non-expandable 26
General: Added new
\texttt{\perlnewenvironment} and
\texttt{\perlrereenvironment}
macros 21

v1.2
\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
Perl\TeX\ macro invocations. 26
General: Renamed
\texttt{perlmacros.sty} to
\texttt{perltex.sty} for consistency 1

v1.3
\texttt{\perldo}: Introduced \texttt{\perldo} to
support code execution outside
of all subroutines 23
\texttt{\plmac@run@code}: Added to assist
\texttt{\perldo} 23
General: Modified \texttt{perltex.pl} to
eschew the sandbox altogether
when \texttt{--nosafe} is specified 28
v1.4
General: Added support for a
--makesty option that
generates a PerlTeX-free style
file called noperltex.sty .... 38

v1.5
\plmac@file@existsfalse:
Modified to read from a named
pipe before checking file
existence ..................... 25

v1.6
General: Added an
(undocumented) --nopipe
option to perltex.pl to help it
work with X\TeX ............. 31

v1.7
$pipestring: Introduced this
variable as a workaround for
X\TeX's attempt to rewind
$pipewillbeused. ................... 30
General: Added an
(undocumented) --synctext
option to alter the text written
to $pipewillbeused ............. 31

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/Perltex
documents ................... 26

v1.9
\plmac@await@existence: Put the
\input\plmac@pipe within an
lrbox environment to prevent
a partial read from introducing
spurious text into the
document ..................... 25
General: Introduced handlers for
SIGALRM and SIGPIPE to make
perltex.pl more robust to
latex exiting at an
inopportune time ............. 32
$awaitexists: Hoisted
$awaitexists from the main
loop and made it a top-level
 subroutine .................... 33
delete_files: Replaced all
unlink...while ~e statements
with calls to a new
delete_files subroutine ...... 32

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

v2.1
\otherstuff: Normalized line
endings across
Unix/Windows/Macintosh .... 35
General: Replaced \abs_path() with File::Spec->rel2abs()
because the latter seems to be
more robust to nonexistent files 32

v2.2
perlfalse: Let-bind \plmac@tag
to \relax if \plmac@tag is
undefined. This corrects a
problem when noperltex is
used with newer versions of
\LaTeX ......................... 14

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

\% ............................... 11X, 54X, 368
\$ ............................... 372–375

42