

PerlTeX: Defining L^AT_EX macros in terms of Perl code*

Scott Pakin
scott+pt@pakin.org

December 10, 2007

Abstract

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

1 Introduction

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

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

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

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

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

*This document corresponds to PerlTeX v1.7, dated 2007/12/10.

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 L^AT_EX to extract a substring of a given string when provided with a starting position and a length. Perl has an built-in `substr` function and Perl_T_EX makes it easy to export this to L^AT_EX:

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

`\substr` can then be used just like any other L^AT_EX macro—and as simply as Perl’s `substr` function:

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



A sample substring of “superlative” is “perl”.

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

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

\hilbertmatrix{20}
```



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

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

```

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

```

```

    }
    return $tabular;
  }{
    return "\\end{tabular}\\n";
  }

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

```



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

2 Usage

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

2.1 Defining and redefining Perl macros

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

A PerlTeX-defined macro or environments is converted to a Perl subroutine named after the macro/environment but beginning with “`latex_`”. For example, a PerlTeX-defined L^AT_EX macro called `\myMacro` internally produces a Perl

subroutine called `latex_myMacro`. Macro arguments are converted to subroutine arguments. A \LaTeX macro's #1 argument is referred to as `$_[0]` in Perl; #2 is referred to as `$_[1]`; and so forth.

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

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

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



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

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

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

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

2.2 Invoking `perltex.pl`

The following pages reproduce the `perltex.pl` program documentation. Key parts of the documentation are excerpted when `perltex.pl` is invoked with the `--help` option. The various Perl `pod2(something)` tools can be used to generate the

complete program documentation in a variety of formats such as L^AT_EX, HTML, plain text, or Unix man-page format. For example, the following command is the recommended way to produce a Unix man page from `perltex.pl`:

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

NAME

perltex — enable L^AT_EX macros to be defined in terms of Perl code

SYNOPSIS

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

DESCRIPTION

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

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

OPTIONS

perltex accepts the following command-line options:

--help

Display basic usage information.

--latex=program

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

--[no]safe

Enable or disable sandboxing. With the default of **--safe**, **perltex** executes the code from a `\perlnewcommand` or `\perlrenewcommand` macro within a protected environment that prohibits “unsafe” operations such as accessing files or executing external programs. Specifying **--nosafe** gives the L^AT_EX 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 L^AT_EX style file called *noperltex.sty*. Replacing the document's `\usepackage{perltex}` line with `\usepackage{noperltex}` produces the same output but does not require PerlT_EX, making the document suitable for distribution to people who do not have PerlT_EX installed. The disadvantage is that *noperltex.sty* is specific to the document that produced it. Any changes to the document's PerlT_EX 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 L^AT_EX 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=:fileysys_open  
--permit=time myfile.tex
```

ENVIRONMENT

perltex honors the following environment variables:

PERLTEX

Specify the filename of the L^AT_EX compiler. The L^AT_EX compiler defaults to “**latex**”. The **PERLTEX** environment variable overrides this default, and the **--latex** command-line option (see **OPTIONS**) overrides that.

FILES

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

jobname.lgpl

log file written by Perl; helpful for debugging Perl macros

jobname.topl

information sent from L^AT_EX to Perl

jobname.frpl

information sent from Perl to L^AT_EX

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 PerlT_EX macro invocation

NOTES

perltex’s sandbox defaults to what *Opcodes* calls “:browse”.

SEE ALSO

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

AUTHOR

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

3 Implementation

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

Section 3 is split into two main parts. Section 3.1 presents the source code for `perltex.sty`, the L_AT_EX side of Perl_T_EX, and Section 3.2 presents the source code for `perltex.pl`, the Perl side of Perl_T_EX. In toto, Perl_T_EX consists of a relatively small amount of code. `perltex.sty` is only 226 lines of L_AT_EX and `perltex.pl` is only 302 lines of Perl. `perltex.pl` is fairly straightforward Perl code and shouldn't be too difficult to understand by anyone comfortable with Perl programming. `perltex.sty`, in contrast, contains a bit of L_AT_EX trickery and is probably impenetrable to anyone who hasn't already tried his hand at L_AT_EX programming. Fortunately for the reader, the code is profusely commented so the aspiring L_AT_EX guru may yet learn something from it.

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

3.1 `perltex.sty`

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

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

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

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

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

3.1.1 Package initialization

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

Table 1: Variables used for communication between Perl and L^AT_EX

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

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

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

```
1 \newif\ifplmac@have@perltx
2 \plmac@have@perltxtrue
3 \@ifundefined{plmac@tag}{\plmac@have@perltxfalse}{}
4 \@ifundefined{plmac@tofile}{\plmac@have@perltxfalse}{}
5 \@ifundefined{plmac@fromfile}{\plmac@have@perltxfalse}{}
6 \@ifundefined{plmac@toflag}{\plmac@have@perltxfalse}{}
7 \@ifundefined{plmac@fromflag}{\plmac@have@perltxfalse}{}
8 \@ifundefined{plmac@doneflag}{\plmac@have@perltxfalse}{}
9 \@ifundefined{plmac@pipe}{\plmac@have@perltxfalse}{}
10 \ifplmac@have@perltx
11 \else
12   \PackageError{perltx}{Document must be compiled using perltx}
13   {Instead of compiling your document directly with latex, you need
14    to\MessageBreak use the perltx script. \space perltx sets up
15    a variety of macros needed by\MessageBreak the perltx
16    package as well as a listener process needed for\MessageBreak
17    communication between LaTeX and Perl.}
18 \fi
```

3.1.2 Defining Perl macros

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

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

1. The user invokes `\perl[re]newcommand`, which stores `\[re]newcommand` in `\plmac@command`. The `\perl[re]newcommand` macro then invokes `\plmac@newcommand@i` with a first argument of “*” for `\perl[re]newcommand*` or “!” for ordinary `\perl[re]newcommand`.
2. `\plmac@newcommand@i` defines `\plmac@starchar` as “*” if it was passed a “*” or *empty* if it was passed a “!”. It then stores the name of the user’s macro in `\plmac@macname`, a `\writeable` version of the name in `\plmac@cleaned@macname`, and the macro’s previous definition (needed by `\perlrenewcommand`) in `\plmac@oldbody`. Finally, `\plmac@newcommand@i` invokes `\plmac@newcommand@ii`.
3. `\plmac@newcommand@ii` stores the number of arguments to the user’s macro (which may be zero) in `\plmac@numargs`. It then invokes `\plmac@newcommand@iii@opt` if the first argument is supposed to be optional or `\plmac@newcommand@iii@no@opt` if all arguments are supposed to be required.
4. `\plmac@newcommand@iii@opt` defines `\plmac@defarg` as the default value of the optional argument. `\plmac@newcommand@iii@no@opt` defines it as *empty*. Both functions then call `\plmac@haveargs`.
5. `\plmac@haveargs` stores the user’s macro body (written in Perl) verbatim in `\plmac@perlcode`. `\plmac@haveargs` then invokes `\plmac@havecode`.
6. By the time `\plmac@havecode` is invoked all of the information needed to define the user’s macro is available. Before defining a L^AT_EX macro, however, `\plmac@havecode` invokes `\plmac@write@perl` to tell `perltex.pl` to define a Perl subroutine with a name based on `\plmac@cleaned@macname` and the code contained in `\plmac@perlcode`. Figure 1 illustrates the data that `\plmac@write@perl` passes to `perltex.pl`.
7. `\plmac@havecode` invokes `\newcommand` or `\renewcommand`, as appropriate, defining the user’s macro as a call to `\plmac@write@perl`. An invocation of the user’s L^AT_EX macro causes `\plmac@write@perl` to pass the information shown in Figure 2 to `perltex.pl`.

DEF
\plmac@tag
\plmac@cleaned@macname
\plmac@tag
\plmac@perlcode

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

USE
\plmac@tag
\plmac@cleaned@macname
\plmac@tag
#1
\plmac@tag
#2
\plmac@tag
#3
:
<i>last</i>

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

8. Whenever \plmac@write@perl is invoked it writes its argument verbatim to \plmac@tofile; perl_{tex}.pl evaluates the code and writes \plmac@fromfile; finally, \plmac@write@perl \inputs \plmac@fromfile.

An example might help distinguish the myriad macros used internally by perl_{tex}.sty. Consider the following call made by the user's document:

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

Table 2 shows how perl_{tex}.sty parses that command into its constituent components and which components are bound to which perl_{tex}.sty macros.

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

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

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

```

19 \def\perlnewcommand{%
20   \let\plmac@command=\newcommand
21   \let\plmac@next=\relax
22   \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
23 }

24 \def\perlrenewcommand{%
25   \let\plmac@next=\relax
26   \let\plmac@command=\renewcommand
27   \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
28 }

```

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

```

29 \def\plmac@newcommand@i#1#2{%
30   \ifx#1*%
31     \def\plmac@starchar{*}%
32   \else
33     \def\plmac@starchar{!}%
34   \fi
35   \def\plmac@macname{#2}%
36   \let\plmac@oldbody=#2\relax
37   \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
38     \expandafter\string\plmac@macname}%
39   \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%
40 }

```

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

```

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

```

```

43 \@ifnextchar[{\plmac@newcommand@iii@opt}
44             {\plmac@newcommand@iii@no@opt}%]
45 }

```

```

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

```

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

```

46 \def\plmac@newcommand@iii@opt[#1]{%
47   \def\plmac@defarg{#1}%
48   \plmac@haveargs
49 }

50 \def\plmac@newcommand@iii@no@opt{%
51   \let\plmac@defarg=\relax
52   \plmac@haveargs
53 }

```

```

\plmac@perlcode
\plmac@haveargs

```

Now things start to get tricky. We have all of the arguments we need to define the user’s command so all that’s left is to grab the macro body. But there’s a catch: Valid Perl code is unlikely to be valid L^AT_EX 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.

```

54 \newtoks\plmac@perlcode

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

```

65 }

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

<code>\plmac@sep</code>	Define a character to separate each of the items presented in Figures 1 and 2. Perl will need to strip this off each argument. For convenience in porting to languages with less powerful string manipulation than Perl's, we define <code>\plmac@sep</code> as a carriage-return character of category code 11 ("letter"). 66 { <code>\catcode'\^^M=11\gdef\plmac@sep{^^M}</code> }
<code>\plmac@argnum</code>	Define a loop variable that will iterate from 1 to the number of arguments in the user's function, i.e., <code>\plmac@numargs</code> . 67 <code>\newcount\plmac@argnum</code>
<code>\plmac@havecode</code>	Now comes the final piece of what started as a call to <code>\perl[re]newcommand</code> . First, to reset all category codes back to normal, <code>\plmac@havecode</code> ends the group that was begun in <code>\plmac@haveargs</code> . 68 <code>\def\plmac@havecode{%</code> 69 <code>\endgroup</code>
<code>\plmac@define@sub</code>	We invoke <code>\plmac@write@perl</code> to define a Perl subroutine named after <code>\plmac@cleaned@macname</code> . <code>\plmac@define@sub</code> sends Perl the information shown in Figure 1 on page 13. 70 <code>\edef\plmac@define@sub{%</code> 71 <code>\noexpand\plmac@write@perl{DEF\plmac@sep</code> 72 <code>\plmac@tag\plmac@sep</code> 73 <code>\plmac@cleaned@macname\plmac@sep</code> 74 <code>\plmac@tag\plmac@sep</code> 75 <code>\the\plmac@perlcode</code> 76 <code>}%</code> 77 <code>}%</code> 78 <code>\plmac@define@sub</code>
<code>\plmac@body</code>	The rest of <code>\plmac@havecode</code> is preparation for defining the user's macro. (\LaTeX 2_ϵ 's <code>\newcommand</code> or <code>\renewcommand</code> will do the actual work, though.) <code>\plmac@body</code> 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 13 (with intervening <code>\plmac@seps</code>). 79 <code>\edef\plmac@body{%</code> 80 <code>USE\plmac@sep</code> 81 <code>\plmac@tag\plmac@sep</code> 82 <code>\plmac@cleaned@macname</code> 83 <code>}%</code>

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

```

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

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

```

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

The final steps are to restore the previous definition of the user’s macro—we had set it to `\relax` above to make the name unexpandable—then redefine it

by invoking `\plmac@define@command`. Why do we need to restore the previous definition if we’re just going to redefine it? Because `\newcommand` needs to produce an error if the macro was previously defined and `\renewcommand` needs to produce an error if the macro was *not* previously defined.

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

```
111 \expandafter\let\plmac@macname=\plmac@oldbody
112 \plmac@define@command
113 \plmac@next
114 }
```

3.1.3 Defining Perl environments

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

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

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

```
115 \def\perlnewenvironment{%
116   \let\plmac@command=\newcommand
117   \let\plmac@next=\plmac@end@environment
118   \ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
119 }

120 \def\perlrenewenvironment{%
121   \let\plmac@command=\renewcommand
122   \let\plmac@next=\plmac@end@environment
123   \ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
124 }
```

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

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

```

125 \def\plmac@newenvironment@i#1#2{%
126   \ifx#1*%
127     \def\plmac@starchar{*}%
128   \else
129     \def\plmac@starchar{}%
130   \fi
131   \def\plmac@envname{#2}%
132   \expandafter\def\expandafter\plmac@macname\expandafter{\csname#2\endcsname}%
133   \expandafter\let\expandafter\plmac@oldbody\plmac@macname\relax
134   \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
135     \expandafter\string\plmac@macname}%
136   \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
137 }

```

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

`\plmac@end@environment`'s purpose is to define `\end<name>`. This is a little tricky, however, because L^AT_EX'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`.

```

138 \def\plmac@end@environment{%
139   \expandafter\def\expandafter\plmac@next\expandafter{\expandafter
140     \let\csname end\plmac@envname\endcsname=\plmac@end@macro
141     \let\plmac@next=\relax
142   }%
143   \def\plmac@macname{\plmac@end@macro}%
144   \expandafter\let\expandafter\plmac@oldbody\csname end\plmac@envname\endcsname
145   \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
146     \expandafter\string\plmac@macname}%
147   \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
148 }

```

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_TE_X macro, `\perldo`, instructs the Perl

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

RUN
\plmac@tag
<i>Ignored</i>
\plmac@tag
\plmac@perlcode

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

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

```

149 \def\perldo{%
150   \begingroup
151   \let\do\@makeother\dospecials
152   \catcode'\^^M=\active
153   \newlinechar'\^^M
154   \endlinechar='\^^M
155   \catcode'\{=1
156   \catcode'\}=2
157   \afterassignment\plmac@have@run@code
158   \global\plmac@perlcode
159 }
```

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

```

160 \def\plmac@have@run@code{%
161   \endgroup
162   \edef\plmac@run@code{%
163     \noexpand\plmac@write@perl{RUN\plmac@sep
164       \plmac@tag\plmac@sep
165       N/A\plmac@sep
166       \plmac@tag\plmac@sep
167       \the\plmac@perlcode
168     }%
169   }%
170   \plmac@run@code
171 }
```

3.1.5 Communication between L^AT_EX and Perl

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

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

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

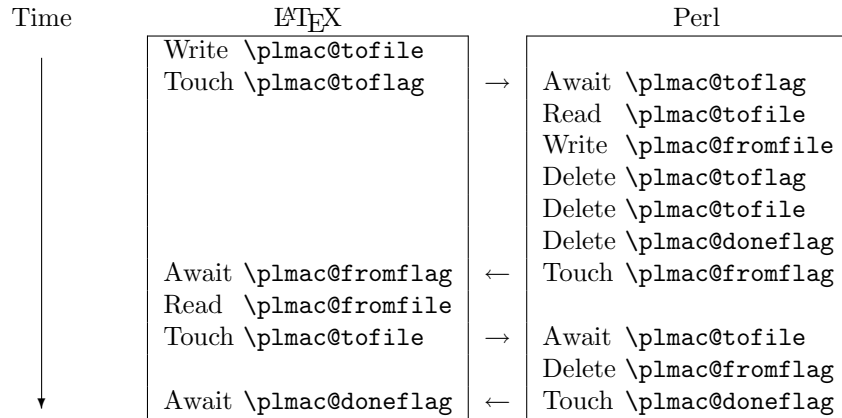


Figure 4: L^AT_EX-to-Perl communication protocol

`\plmac@await@existence` The purpose of the `\plmac@await@existence` macro is to repeatedly check
`\ifplmac@file@exists` the existence of a given file until the file actually exists. For conven-
`\plmac@file@existstrue` nience, we use L^AT_EX 2_ε’s `\IfFileExists` macro to check the file and invoke
`\plmac@file@existsfalse` `\plmac@file@existstrue` or `\plmac@file@existsfalse`, as appropriate.

As a performance optimization we `\input` a named pipe. This causes the `latex` process to relinquish the CPU until the `perltex` process writes data (always just “`\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 “\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 L^AT_EX and Perl.

```

172 \newif\ifplmac@file@exists
173 \newcommand{\plmac@await@existence}[1]{%
174   \input\plmac@pipe
175   \loop
176     \IfFileExists{#1}%
177       {\plmac@file@existstrue}%
178       {\plmac@file@existfalse}%
179   \ifplmac@file@exists
180   \else
181   \repeat
182 }
```

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

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

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

```

184 \newcommand{\plmac@write@perl}{%
185   \begingroup
186   \let\do\@makeother\dospecials
187   \catcode'\^M=\active
188   \newlinechar'\^M
189   \endlinechar='\^M
190   \catcode'\{=1
191   \catcode'\}=2
192   \plmac@write@perl@i
193 }
```

\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 L^AT_EX 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 (L^AT_EX) result.

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

```

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

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

```

196      \immediate\openout\plmac@outfile=\plmac@tofile\relax
197      \let\protect=\noexpand
198      \def\begin{\noexpand\begin}%
199      \def\end{\noexpand\end}%
200      \immediate\write\plmac@outfile{#1}%
201      \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.

```

202      \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`.

```

203      \immediate\openout\plmac@outfile=\plmac@toflag\relax
204      \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`.

```

205      \plmac@await@existence\plmac@fromflag

```

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

Because T_EX can’t delete files we require an additional L^AT_EX-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`.

```

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

```

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

```

209      \input\plmac@fromfile\relax
210  }

```

`\plmac@write@perl@i` The foregoing code represents the “real” definition of `\plmac@write@perl@i`. For the user’s convenience, we define a dummy version of `\plmac@write@perl@i` so that a document which utilizes `perltex.sty` can still compile even if not built using `perltex.pl`. All calls to macros defined with `\perl[re]newcommand` and all invocations of environments defined with `\perl[re]newenvironment` are replaced with “PerlT_EX”. A minor complication is that text can’t be inserted before the

`\begin{document}`}. Hence, we initially define `\plmac@write@perl@i` as a doing nothing macro and redefine it as “`\fbox{Perl\TeX}`” at the `\begin{document}`.

```
211 \else
212   \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.

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

221 \AtBeginDocument{%
222   \renewcommand{\plmac@write@perl@i}[1]{%
223     \plmac@show@placeholder#1\@empty
224   }%
225 }
226 \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.

```
227 #! /usr/bin/env perl
228
229 #####
230 # Prepare a LaTeX run for two-way communication with Perl #
231 # By Scott Pakin <scott+pt@pakin.org> #
232 #####
233
234 #-----
235 # This is file 'perltex.pl',
236 # generated with the docstrip utility.
```



```

237 #
238 # The original source files were:
239 #
240 # perltex.dtx (with options: 'perltex')
241 #
242 # This is a generated file.
243 #
244 # Copyright (C) 2007 Scott Pakin <scott+pt@pakin.org>
245 #
246 # This file may be distributed and/or modified under the conditions
247 # of the LaTeX Project Public License, either version 1.3c of this
248 # license or (at your option) any later version. The latest
249 # version of this license is in:
250 #
251 #   http://www.latex-project.org/lppl.txt
252 #
253 # and version 1.3c or later is part of all distributions of LaTeX
254 # version 2006/05/20 or later.
255 #-----
256

```

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.

```

257 sub top_level_eval ($)
258 {
259     return eval $_[0];
260 }

```

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.

```

261 use Safe;
262 use Opcode;
263 use Getopt::Long;
264 use Pod::Usage;
265 use File::Basename;
266 use Fcntl;
267 use POSIX;
268 use warnings;

```

```
269 use strict;
```

3.2.4 Variable declarations

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

Variables corresponding to command-line arguments

<code>\$latexprog</code>	<code>\$latexprog</code> is the name of the L ^A T _E X executable (e.g., “ <code>latex</code> ”).
<code>\$runsafely</code>	If <code>\$runsafely</code> is 1 (the default), then the user’s Perl code runs in a secure sandbox; if it’s 0, then arbitrary Perl code is allowed to run.
<code>@permittedops</code>	<code>@permittedops</code> is a list of features made available to the user’s Perl code. Valid values are described in Perl’s <code>Opcode</code> manual page.
<code>\$usepipe</code>	<code>perl_{tex}.pl</code> ’s default is a list containing only <code>:browse</code> . <code>\$usepipe</code> is 1 if <code>perl_{tex}.pl</code> should attempt to use a named pipe for communicating with <code>latex</code> or 0 if an ordinary file should be used instead.

```
270 my $latexprog;  
271 my $runsafely = 1;  
272 my @permittedops;  
273 my $usepipe = 1;
```

Filename variables

<code>\$progname</code>	<code>\$progname</code> is the run-time name of the <code>perl_{tex}.pl</code> program.
<code>\$jobname</code>	<code>\$jobname</code> is the base name of the user’s <code>.tex</code> file, which defaults to the T _E X default of <code>texput</code> .
<code>\$toperl</code>	<code>\$toperl</code> defines the filename used for L ^A T _E X-to-Perl communication.
<code>\$fromperl</code>	<code>\$fromperl</code> defines the filename used for Perl-to-L ^A T _E X communication.
<code>\$toflag</code>	<code>\$toflag</code> is the name of a file that will exist only after L ^A T _E X creates <code>\$tofile</code> .
<code>\$fromflag</code>	<code>\$fromflag</code> is the name of a file that will exist only after Perl creates <code>\$fromfile</code> .
<code>\$doneflag</code>	<code>\$doneflag</code> is the name of a file that will exist only after Perl deletes <code>\$fromflag</code> .
<code>\$logfile</code>	<code>\$logfile</code> is the name of a log file to which <code>perl_{tex}.pl</code> writes verbose execution information.
<code>\$pipe</code>	<code>\$pipe</code> 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 <code>\$usepipe</code> is set to 0) used to convince the <code>latex</code> process to yield control of the CPU.

```
274 my $progname = basename $0;  
275 my $jobname = "texput";  
276 my $toperl;  
277 my $fromperl;  
278 my $toflag;  
279 my $fromflag;  
280 my $doneflag;  
281 my $logfile;  
282 my $pipe;
```

Other global variables

@latexcmdline @latexcmdline is the command line to pass to the L^AT_EX executable. **\$styfile** is the string `noperltex.sty` if `perltex.pl` is run with `--makesty`, otherwise undefined. **@macroexpansions** is a list of PerlT_EX macro expansions in the order they were encountered. It is used for creating a `noperltex.sty` file when `--makesty` is specified. **\$sandbox** is a secure sandbox in which to run code that appeared in the L^AT_EX 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.

```
283 my @latexcmdline;  
284 my $styfile;  
285 my @macroexpansions;  
286 my $sandbox = new Safe;  
287 my $sandbox_eval;  
288 my $latexpid;
```

\$pipestring **\$pipestring** is a constant string to write to the **\$pipe** named pipe (or file) at each L^AT_EX synchronization point. Its particular definition is really a bug workaround for X_YT_EX. The current version of X_YT_EX 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 PerlT_EX 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.

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

3.2.5 Command-line conversion

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

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

```
290 $latexprog = $ENV{"PERLTEX"} || "latex";  
291 Getopt::Long::Configure("require_order", "pass_through");  
292 GetOptions("help" => sub {pod2usage(-verbose => 1)}),  
293           "latex=s" => \$latexprog,  
294           "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.

```

295         "pipe!"      => \usepipe,
296         "synctext=s" => \pipestring,
297         "makesty"    => sub {$styfile = "noperltex.sty"},
298         "permit=s"   => \@permittedops) || pod2usage(2);

```

Preparing a L^AT_EX command line

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

```

299 @latexcmdline = @ARGV;
300 my $firstcmd = 0;
301 for ($firstcmd=0; $firstcmd<=$#latexcmdline; $firstcmd++) {
302     my $option = $latexcmdline[$firstcmd];
303     next if substr($option, 0, 1) eq "-";
304     if (substr ($option, 0, 1) ne "\\") {
305         $jobname = basename $option, ".tex" ;
306         $latexcmdline[$firstcmd] = "\\input $option";
307     }
308     last;
309 }
310 push @latexcmdline, "" if $#latexcmdline==-1;

```

\$separator To avoid conflicts with the code and parameters passed to Perl from L^AT_EX (see Figure 1 on page 13 and Figure 2 on page 13) we define a separator string, **\$separator**, containing 20 random uppercase letters.

```

311 my $separator = "";
312 foreach (1 .. 20) {
313     $separator .= chr(ord("A") + rand(26));
314 }

```

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

```

315 $toperl = $jobname . ".topl";
316 $fromperl = $jobname . ".frpl";
317 $toflag = $jobname . ".tfpl";
318 $fromflag = $jobname . ".ffpl";
319 $doneflag = $jobname . ".dfpl";
320 $logfile = $jobname . ".lgpl";
321 $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.

```

322 $latexcmdline[$firstcmd] =
323     sprintf '\makeatletter' . '\def%s{%s}' x 7 . '\makeatother%s',
324     '\plmac@tag', $separator,
325     '\plmac@tofile', $toperl,

```

```

326     '\plmac@fromfile', $fromperl,
327     '\plmac@toflag', $toflag,
328     '\plmac@fromflag', $fromflag,
329     '\plmac@doneflag', $doneflag,
330     '\plmac@pipe', $pipe,
331     $latexcmdline[$firstcmd];

```

3.2.6 Launching L^AT_EX

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 L^AT_EX 2_ε 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.

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

```

332 foreach my $file ($toperl, $fromperl, $toflag, $fromflag, $doneflag, $pipe) {
333     unlink $file while -e $file;
334 }
335 open (LOGFILE, ">$logfile") || die "open(\"$logfile\"): $!\n";
336 if (defined $styfile) {
337     open (STYFILE, ">$styfile") || die "open(\"$styfile\"): $!\n";
338 }
339 if (!$usepipe || !eval {mkfifo($pipe, 0600)}) {
340     sysopen PIPE, $pipe, O_WRONLY|O_CREAT, 0755;
341     print PIPE $pipestring;
342     close PIPE;
343     $usepipe = 0;
344 }
345 defined ($latexpid = fork) || die "fork: $!\n";
346 unshift @latexcmdline, $latexprog;
347 if (!$latexpid) {
348     exec {@latexcmdline[0]} @latexcmdline;
349     die "exec('@latexcmdline'): $!\n";
350 }

```

3.2.7 Preparing a sandbox

`perltex.pl` uses Perl's `Safe` and `Opcode` modules to declare a secure sandbox (`$sandbox`) in which to run Perl code passed to it from L^AT_EX. 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.

```

351 if ($runsafely) {
352     @permittedops=(":browse") if $#permittedops==1;
353     $sandbox->permit_only (@permittedops);
354     $sandbox_eval = sub {$sandbox->reval($_[0])};
355 }
356 else {
357     $sandbox_eval = \&top_level_eval;
358 }

```

3.2.8 Communicating with L^AT_EX

The following code constitutes `perltex.pl`'s main loop. Until `latex` exits, the loop repeatedly reads Perl code from L^AT_EX, evaluates it, and returns the result as per the protocol described in Figure 4 on page 21.

```

359 while (1) {

```

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

```

360     my $awaitexists = sub {
361         while (!-e $_[0]) {
362             sleep 0;
363             if (waitpid($latexpid, &WNOHANG)==-1) {
364                 foreach my $file ($topperl, $fromperl, $toflag,
365                     $fromflag, $doneflag, $pipe) {
366                     unlink $file while -e $file;
367                 }
368                 undef $latexpid;
369                 exit 0;
370             }
371         }
372     };

```

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

```

373     $awaitexists->($toflag);
374     my $entirefile;
375     {
376         local $/ = undef;
377         open (TOPERL, "<$topperl") || die "open($topperl): !\n";
378         $entirefile = <TOPERL>;
379         close TOPERL;
380     }

```

\$optag 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 `@otherstuff`

DEF then `@otherstuff` will contain the Perl code to define. If `$optag` is USE then `@otherstuff` will be a list of subroutine arguments. If `$optag` is RUN then `@otherstuff` will be a block of Perl code to run.

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

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

```
383     $macroname =~ s/^[^A-Za-z]+//;
384     $macroname =~ s/\W/_/g;
385     $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 “\,”.

```
386     if ($optag eq "USE") {
387         foreach (@otherstuff) {
388             s/\\/\\\\/g;
389             s/\'/\\\'/g;
390             $_ = "\$_";
391         }
392     }
```

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

```
393     my $perlcode;
394     if ($optag eq "DEF") {
395         $perlcode =
396             sprintf "sub %s {%s}\n",
397                 $macroname, $otherstuff[0];
398     }
399     elsif ($optag eq "USE") {
400         $perlcode = sprintf "%s (%s);\n", $macroname, join(" ", @otherstuff);
401     }
402     elsif ($optag eq "RUN") {
403         $perlcode = $otherstuff[0];
404     }
405     else {
406         die "${programe}: Internal error -- unexpected operation tag \"$optag\"\n";
407     }
```

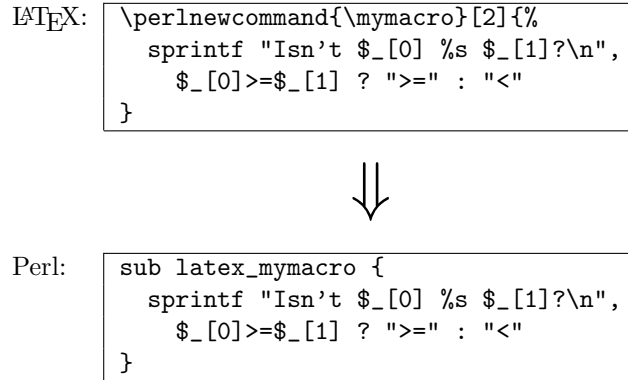


Figure 5: Conversion from L^AT_EX to Perl (subroutine definition)

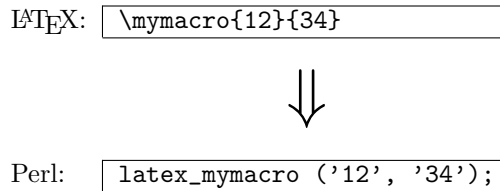


Figure 6: Conversion from L^AT_EX to Perl (subroutine invocation)

Log what we're about to evaluate.

```
408     print LOGFILE "#" x 31, " PERL CODE ", "#" x 32, "\\n";
409     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., `$@` is defined) we replace the result (**\$result**) with a call to L^AT_EX 2_ε's `\PackageError` macro to return a suitable error message. We produce one error message for sandbox policy violations (detected by the error message, `$@`, 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 L^AT_EX code or as a `\PackageError`—we store it in `@macroexpansions` in preparation for writing it to `noperltex.sty` (when `perltex.pl` is run with `--makesty`).

```
410     undef $_;
411     my $result;
412     {
413         my $warningmsg;
414         local $SIG{__WARN__} =
415             sub {chomp ($warningmsg=$_[0]); return 0};
416         $result = $sandbox_eval->($perlcode);
417         if (defined $warningmsg) {
```



```

418         $warningmsg =~ s/at \(\eval \d+\) line \d+\W+//;
419         print LOGFILE "# ==> $warningmsg\n\n";
420     }
421 }
422 $result = "" if !$result || $optag eq "RUN";
423 if ($?) {
424     my $msg = $?;
425     $msg =~ s/at \(\eval \d+\) line \d+\W+//;
426     $msg =~ s/\s+ / /;
427     $result = "\\PackageError{perltx}{$msg}";
428     my @helpstring;
429     if ($msg =~ /\btrapped by\b/) {
430         @helpstring =
431             ("The preceding error message comes from Perl. Apparently,",
432              "the Perl code you tried to execute attempted to perform an",
433              "'unsafe' operation. If you trust the Perl code (e.g., if",
434              "you wrote it) then you can invoke perltx with the --nosafe",
435              "option to allow arbitrary Perl code to execute.",
436              "Alternatively, you can selectively enable Perl features",
437              "using perltx's --permit option. Don't do this if you don't",
438              "trust the Perl code, however; malicious Perl code can do a",
439              "world of harm to your computer system.");
440     }
441     else {
442         @helpstring =
443             ("The preceding error message comes from Perl. Apparently,",
444              "there's a bug in your Perl code. You'll need to sort that",
445              "out in your document and re-run perltx.");
446     }
447     my $helpstring = join ("\\MessageBreak\n", @helpstring);
448     $helpstring =~ s/\. /\.\.\space\space /g;
449     $result .= "{$helpstring}";
450 }
451 push @macroexpansions, $result if defined $styfile && $optag eq "USE";

```

Log the resulting L^AT_EX code.

```

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

```

We add `\endinput` to the generated L^AT_EX code to suppress an extraneous end-of-line character that T_EX would otherwise insert.

```

454 $result .= '\endinput';

```

Continuing the protocol described in Figure 4 on page 21 we now write `$result` (which contains either the result of executing the user's or a `\PackageError`) to the `$fromperl` file, delete `$toflag`, `$topperl`, and `$doneflag`, and notify L^AT_EX by touching the `$fromflag` file. As a performance optimization, we also write `\endinput` into `$pipe` to wake up the latex process.

```

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

```

```

457     close FROMPERL;
458     unlink $toflag while -e $toflag;
459     unlink $toperl while -e $toperl;
460     unlink $doneflag while -e $doneflag;
461     open (FROMFLAG, ">$fromflag") || die "open($fromflag): !\n";
462     close FROMFLAG;
463     if (open (PIPE, ">$pipe")) {
464         print PIPE $pipestring;
465         close PIPE;
466     }

```

We have to perform one final L^AT_EX-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.

```

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

```

Again, we awaken the latex process, which is blocked on `$pipe`.

```

471     if (open (PIPE, ">$pipe")) {
472         print PIPE $pipestring;
473         close PIPE;
474     }
475 }

```

3.2.9 Final cleanup

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

```

476 END {
477     close LOGFILE;
478     if (defined $latexpid) {
479         kill (9, $latexpid);
480         exit 1;
481     }
482
483     if (defined $styfile) {

```

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

```

484         print STYFILE <<"STYFILEHEADER1";
485         \\NeedsTeXFormat{LaTeX2e}[1999/12/01]
486         \\ProvidesPackage{noperlTEX}
487         [2007/09/29 v1.4 Perl-free version of PerlTEX specific to $jobname.tex]
488 STYFILEHEADER1
489         ;

```

```

490         print STYFILE <<'STYFILEHEADER2';
491 \RequirePackage{filecontents}
492
493 % Suppress the "Document must be compiled using perltex" error from perltex.
494 \let\noperltex@PackageError=\PackageError
495 \renewcommand{\PackageError}[3]{%
496 \RequirePackage{perltex}
497 \let\PackageError=\noperltex@PackageError
498

```

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

```

499 % Modify \plmac@show@placeholder to input the next noperltex-*.tex file
500 % each time a PerlTeX-defined macro is invoked.
501 \newcount\plmac@macro@invocation@num
502 \gdef\plmac@show@placeholder#1#2\@empty{%
503   \ifx#1U\relax
504     \endgroup
505     \advance\plmac@macro@invocation@num by 1\relax
506     \global\plmac@macro@invocation@num=\plmac@macro@invocation@num
507     \input{noperltex-\the\plmac@macro@invocation@num.tex}%
508   \else
509     \endgroup
510   \fi
511 }
512 STYFILEHEADER2
513   ;

```

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.

```

514   foreach my $e (0 .. $#macroexpansions) {
515     print STYFILE "\n";
516     printf STYFILE "%% Invocation %#d\n", 1+$e;
517     printf STYFILE "\\begin{filecontents}{noperltex-%d.tex}\n", 1+$e;
518     print STYFILE $macroexpansions[$e], "\\endinput\n";
519     print STYFILE "\\end{filecontents}\n";
520   }
521   print STYFILE "\\endinput\n";
522   close STYFILE;
523 }
524
525   exit 0;
526 }
527

```

3.2.10 perl_{tex}.pl POD documentation

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

3.3 Porting to other languages

Perl is a natural choice for a L^AT_EX macro language because of its excellent support for text manipulation including extended regular expressions, string interpolation, and “here” strings, to name a few nice features. However, Perl’s syntax is unusual and its semantics are rife with annoying special cases. Some users will therefore long for a *⟨some-language-other-than-Perl⟩*T_EX. Fortunately, porting PerlT_EX 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 “PerlT_EX”) as per the PerlT_EX license agreement (Section 4).
- In your replacement for `perltex.sty`, replace all occurrences of “`plmac`” with a different string.
- In your replacement for `perltex.pl`, choose different file extensions for the various helper files.

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

4 License agreement

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

These files may be distributed and/or modified under the conditions of the L^AT_EX Project Public License, either version 1.3c of this license or (at your option) any later version. The latest version of this license is in <http://www.latex-project.org/lppl.txt> and version 1.3c or later is part of all distributions of L^AT_EX 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; and to Nick Andrewes for identifying and helping diagnose a problem running PerlTeX with XeTeX and to Jonathan Kew for suggesting a workaround. 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		in order to support recursive PerlTeX macro invocations. . .	23
General: Initial version	1		
v1.0a			
General: Made all <code>unlink</code> calls wait for the file to actually disappear	29	v1.3 General: Modified <code>perltex.pl</code> to eschew the sandbox altogether when <code>--nosafe</code> is specified	25
Undefined <code>\$/</code> only locally	30	<code>\perldo</code> : Introduced <code>\perldo</code> to support code execution outside of all subroutines.	20
<code>\$awaitexists</code> : Bug fix: Added “ <code>undef \$latexpid</code> ” to make the <code>END</code> block correctly return a status code of 0 on success . .	30	<code>\plmac@run@code</code> : Added to assist <code>\perldo</code>	20
v1.1			
General: Added new <code>\perlnewenvironment</code> and <code>\perlrenewenvironment</code> macros	18	v1.4 General: Added support for a <code>--makesty</code> option that generates a PerlTeX-free style file called <code>noperltex.sty</code>	34
<code>\plmac@havecode</code> : Added a <code>\plmac@next</code> hook to support PerlTeX’s new environment-defining macros	16	v1.5 <code>\plmac@file@existsfalse</code> : Modified to read from a named pipe before checking file existence .	22
<code>\plmac@write@perl@i</code> : Added a dummy version of the macro to use if <code>latex</code> was launched directly, without <code>perltex.pl</code> . .	23	v1.6 General: Added an (undocumented) <code>--npipe</code> option to <code>perltex.pl</code> to help it work with XeTeX	27
Made argument-handling more rational by making <code>\protect</code> , <code>\begin</code> , and <code>\end</code> non-expandable	23	v1.7 General: Added an (undocumented) <code>--synctext</code> option to alter the text written to <code>\$pipe</code>	27
v1.2		<code>\$pipestring</code> : Introduced this variable as a workaround for XeTeX’s attempt to rewind <code>\$pipe</code>	27
General: Renamed <code>perlmacros.sty</code> to <code>perltex.sty</code> for consistency.	1		
<code>\plmac@write@perl@i</code> : Moved the <code>\input</code> of the generated Perl code to the end of the routine			

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		<code>\$latexprog</code> 270
<code>\\$</code> 293–296		<code>\$logfile</code> 274
<code>\%</code> 289		
<code>\&</code> 357		M
<code>\{</code> 61, 155, 190		<code>@macroexpansions</code> 283
<code>\}</code> 62, 156, 191		<code>\$macroname</code> 381
<code>\^</code> 58–60, 66, 152–154, 187–189		<code>\$msg</code> 410
A		N
<code>\afterassignment</code> 63, 157		<code>\newcommand</code> . 20, 116, 173, 184, 195, 212
<code>\AtBeginDocument</code> 221		<code>\newlinechar</code> 59, 153, 188
<code>\$awaitexists</code> 360		<code>\noperltex@PackageError</code> ... 494, 497
C		O
<code>\closeout</code> 201, 204, 207		<code>\openout</code> 196, 203, 206
D		<code>\$optag</code> 381
<code>\do</code> 57, 151, 186		<code>\$option</code> 299
<code>\$doneflag</code> 274		<code>@otherstuff</code> 381
<code>\dospecials</code> 57, 151, 186		P
E		<code>\PackageError</code> 12, 494, 495, 497
<code>\endinput</code> 454		<code>\$perlcode</code> 393
<code>\endlinechar</code> 60, 154, 189		<code>\perldo</code> 149
<code>\$entirefile</code> 373		<code>\perlnewcommand</code> 19
F		<code>\perlnewenvironment</code> 115
<code>\fbox</code> 218		<code>\perlrenewcommand</code> 19
<code>\$firstcmd</code> 299		<code>\perlrenewenvironment</code> 115
<code>\$fromflag</code> 274		<code>@permittedops</code> 270
<code>\$fromperl</code> 274		<code>\$pipe</code> 274
I		<code>\$pipestring</code> 289
<code>\IfFileExists</code> 176		<code>\plmac@argnum</code> 67, 85, 87, 91, 92
<code>\ifplmac@file@exists</code> 172		<code>\plmac@await@existence</code> 172, 205, 208
<code>\ifplmac@have@perltx</code> 1, 194		<code>\plmac@body</code> 79
<code>\input</code> 174, 209, 507		<code>\plmac@cleaned@macname</code>
J	 29, 73, 82, 125, 138
<code>\$jobname</code> 274		<code>\plmac@command</code> 19, 98, 105, 115
L		<code>\plmac@defarg</code> 46, 96, 106
<code>@latexcmdline</code> 283		<code>\plmac@define@command</code> 95
<code>\$latexpid</code> 283		<code>\plmac@define@sub</code> 70
		<code>\plmac@doneflag</code> 208, 329
		<code>\plmac@end@environment</code> 117, 122, 138
		<code>\plmac@end@macro</code> 140, 143
		<code>\plmac@envname</code> 125, 140, 144
		<code>\plmac@file@existsfalse</code> 172

\plmac@file@existstrue	172	\plmac@tag	72, 74, 81, 90, 164, 166, 324
\plmac@fromfile	209, 326	\plmac@tofile	196, 206, 325
\plmac@fromflag	205, 328	\plmac@toflag	203, 327
\plmac@hash	84	\plmac@write@perl	71, 100, 107, 163, 184
\plmac@have@perltextfalse	1	\plmac@write@perl@i	192, 194, 211
\plmac@have@perltexttrue	1	\$progrname	274
\plmac@have@run@code	157, 160		
\plmac@haveargs	48, 52, 54	R	
\plmac@havecode	63, 68	\renewcommand	26, 121, 222, 495
\plmac@macname		\RequirePackage	491, 496
.... 29, 95, 98, 105, 111, 125, 138		\$result	410
\plmac@macro@invocation@num	499	\$runsafely	270
\plmac@newcommand@i	22, 27, 29		
\plmac@newcommand@ii	39, 41, 136, 147	S	
\plmac@newcommand@iii@no@opt	44, 46	\$sandbox	283
\plmac@newcommand@iii@opt	43, 46	\$sandbox_eval	283
\plmac@newenvironment@i	118, 123, 125	\$separator	311
\plmac@next	19, 113, 115, 138	\$styfile	283
\plmac@numargs	41, 87, 99, 106		
\plmac@oldbody	29, 111, 125, 138	T	
\plmac@outfile	183,	\$toflag	274
196, 200, 201, 203, 204, 206, 207		\$toperl	274
\plmac@perlcode	54, 75, 158, 167		
\plmac@pipe	174, 330	U	
\plmac@run@code	160	\$usepipe	270
\plmac@sep	66, 71–74, 80, 81, 90, 163–166		
\plmac@show@placeholder	213, 223, 499	W	
\plmac@starchar	29, 98, 105, 125	\write	200