The tclldoc package and class

Lars Hellström*et al.*

2003/07/19

Abstract

The tclldoc package defines a couple of environments and macros for documenting Tcl source code in `.dtx`-style documented source files, much like what the doc package [8] does for TEX source code. The tclldoc class is analogous to the ltxdoc document class [2]—it loads the package to gain the basic functionality and changes some layout parameters to values that are better suited for documented source than those set by the standard article document class.

The tclldoc package builds on the doc, xdoc [4], and docindex [5] packages.

Note: The tclldoc package and class used to be called tcldoc, but it turned out that there already existed a Perl (!!) script TclDoc that has a similar function. Therefore I added an extra 'l'—which stands for both LATEX and Literate—to avoid confusing the community of Tcl programmers. For compatibility, a package and a class named tcldoc are installed with tclldoc, but they shouldn't be used for new documents.

Contents

*E-mail: Lars.Hellstrom@math.umu.se

1 Introduction

This introduction is meant to be comprehensible even for readers unfamiliar with writing `.dtx` files and using the doc package. Readers who are experienced with this will probably want to skip right to the next section.

A `.dtx` file has a dual nature. On one hand it is a container for some lines of code—it could be a program, a macro package, a configuration file for some program, merely a part of any of the aforementioned, or even arbitrary combinations of the above—and on the other hand it is a LaTeX document which documents this code. One important advantage with this arrangement is that one can keep all the pieces of a project that has to do with a specific task at one place; experience has shown that this greatly furthers really keeping all parts of a project up to date with each other.

Slightly simplified, one can say that a `.dtx` file contains three kinds of lines. A *code line* is a line that doesn't begin with a '`%`' character; such lines can be extracted (copied) using the docstrip program [7]. A *guard line* is a line that begins with the two characters '`%<`'; guards are used to structure the set of code lines so that docstrip can extract different code lines to different generated files. A *comment line*, finally, is a line that begins with a '`%`' character that is not immediately followed by a '`<`' character. The comment lines are ignored by docstrip, but are part of (and usually make up most of) the LaTeX document in the `.dtx` file.

1.1 Special conventions and basic features in a `.dtx` LaTeX document

An important difference between normal LaTeX documents and `.dtx` LaTeX documents is that the percent character '`%`' doesn't start a comment in the latter; in fact it is usually ignored. This allows LaTeX to see and typeset the text in the comment lines of a `.dtx` file. Hence if one wants to include the sentence "Your hovercraft is full of eels!", which in a normal LaTeX document could have been written as the line

```
Your hovercraft is full of eels!
```

one would instead write the line as

```
% Your hovercraft is full of eels!
```

in a `.dtx` document. The space after the `%` is not necessary, but most `.dtx` documents you see include it—probably because the "comment out TeX code" action of most text editors consists of inserting a percent *and* a space at the beginning of each line.

The code lines present the opposite problem, as they usually shouldn't be treated as normal LaTeX code although the normal LaTeX reading conventions would make them the entire document. The usual way around this is to surround each group of code lines with two comment lines that begin and end an environment in which the code lines get typeset verbatim. The tclldoc package provides the `tcl` environment for this purpose, so the code lines

```
proc factorial {n} {
   set result 1
   for {set i 1} {$i<=$n} {incr i} {
```

```
      set result [expr {$result * $i}]
   }
   return $result
}
```

could be included in a `.dtx` document as the lines

```
% \begin{tcl}
proc factorial {n} {
   set result 1
   for {set i 1} {$i<=$n} {incr i} {
      set result [expr {$result * $i}]
   }
   return $result
}
% \end{tcl}
```

When typeset, this will look as

```
1 proc factorial {n} {
2    set result 1
3    for {set i 1} {$i<=$n} {incr i} {
4       set result [expr {$result * $i}]
5    }
6    return $result
7 }
```

The tiny numbers at the beginning of each line enumerate the code lines. Index references to code usually specify such code line numbers, but the enumeration can be switched off.

In mathematical papers, the statements of e.g. theorems are usually made inside a `theorem` (or whatever) environment which provides certain text formatting, a heading, and a position in the document that can be referenced from other parts of it. In `.dtx` documents one usually does something similar for each named piece of code: macros, environments, templates, etc. In particular, the tclldoc package provides two environments `proc` (for procedures) and `variable` (for variables). Figure 1 contains an example of how `proc` might be used in describing a procedure for computing the greatest common divisor of two integers.

What does the `proc` environment do more precisely? First there's the marginal heading which can be seen in Figure 1. Such headings make it easier to find the procedure in the typeset form of the document. Then the `proc` environment makes an index entry which tells you where the procedure is defined, and finally it stores the procedure name in a variable so that subsequent `\changes`[1] commands know to what the change that they are recording was made.

The `variable` environment does the same things except that it writes "(var.)" rather than "(proc)". This environment wasn't used for describing the three local variables `a`, `b`, and `r` in the example; this is since there is no point in referring to these variables from elsewhere in the program. Instead the `variable` environment is primarily meant for global variables (although it could also be useful for local variables that are meant to be accessed using `upvar` or `uplevel`), and as such it can often be of great help, since the description of a global variable can otherwise

[1] The `\changes` command is defined by the `doc` package [8]. It is used for adding entries to a global list of changes for code in the `.dtx` document.

`gcd` (proc) The `gcd` procedure takes two arguments a and b which must be integers and returns their greatest common divisor $\gcd(a, b)$, which is computed using Euclid's algorithm. As a special case, $\gcd(0, 0)$ is considered to be 0, so formally `gcd $a $b` computes $|a| \wedge |b|$, where $\wedge$ denotes the meet operation in the divisor lattice of non-negative integers.

```
 8 proc gcd {a b} {
 9    set a [expr {abs($a)}]
10    set b [expr {abs($b)}]
11    while {$b>0} {
12       set r [expr {$a%$b}]
13       set a $b
14       set b $r
15    }
16    return $a
17 }
```

(a) A typeset procedure with description

```
% \begin{proc}{gcd}
%   The |gcd| procedure takes two arguments $a$ and $b$ which must be
%   integers and returns their greatest common divisor $\gcd(a,b)$,
%   which is computed using Euclid's algorithm. As a special case,
%   $\gcd(0,0)$ is considered to be $0$, so formally |gcd $a $b|
%   computes $\lvert a \rvert \wedge \lvert b \rvert$, where $\wedge$
%   denotes the meet operation in the divisor lattice of non-negative
%   integers.
%   \begin{tcl}
proc gcd {a b} {
   set a [expr {abs($a)}]
   set b [expr {abs($b)}]
   while {$b>0} {
      set r [expr {$a%$b}]
      set a $b
      set b $r
   }
   return $a
}
%   \end{tcl}
% \end{proc}
```

(b) The code for the example in (a)

Figure 1: An example of the `proc` environment

be hard to find, especially with languages like Tcl where variables don't have to be declared and thus have no natural "home" in the code.

Another noteworthy feature in the example is the use of vertical bar '|' characters to delimit short pieces of verbatim Tcl code in the comment lines. It is often necessary for the explanation to include short examples of code in the documentation, and the standard LaTeX `\verb` command is exactly what one would need for this. As such code sections are rather frequent however, it has become the custom to use a single character for both starting and ending such a piece of code. The tclldoc document class defines `|` as a shorthand for `\tclverb|`, where `\tclverb` is a variant of `\verb` which has been designed specifically for Tcl code.

The above description was meant to give a basic understanding of how Tcl code and documentation thereof can be mixed in a `.dtx` file, it neither explains all the environments and commands that the tclldoc package provides, nor mentions all the features of the environments that were described. That information can instead be found in Section 2 of this paper. It should also be mentioned that the doc package [8] defines several commands and environments that may be of use for describing code, and it is well worth getting acquainted with the features of that package as well (although parts of its documentation has become rather archaic).

1.2 Guards and docstrip installation scripts

The central command in a docstrip installation script is `\generate`, since this is the command which actually causes code to be extracted. `\generate`'s syntax is

> `\generate{`⟨*files*⟩`}`

where ⟨*files*⟩ consists of one or several `\file` commands, each of which has the syntax

> `\file{`⟨*output*⟩`}{`⟨*sources*⟩`}`

where ⟨*output*⟩ is a filename and ⟨*sources*⟩ consists of one or several `\from` commands, each of which has the syntax

> `\from{`⟨*input*⟩`}{`⟨*options*⟩`}`

where, finally, ⟨*input*⟩ is a filename and ⟨*options*⟩ is a comma-separated list of alphanumerical strings. Thus a `\generate` command might look like

```
\generate{\file{p1.sty}{\from{s1.dtx}{foo,bar}}
          \file{p2.sty}{\from{s2.dtx}{baz}
                        \from{s3.dtx}{baz}}
          \file{p3.sty}{\from{s1.dtx}{zip}
                        \from{s2.dtx}{zap}}
         }
```

The meaning of this command is

> Generate the three files `p1.sty`, `p2.sty`, and `p3.sty`. Extract the code for `p1.sty` from `s1.dtx` with options `foo` and `bar`, extract the code for `p2.sty` from `s2.dtx` with option `baz` and `s3.dtx` (the code from `s2.dtx` will be put before the code from `s3.dtx`) with option `baz`, and finally extract the code for `p3.sty` from `s1.dtx` with option `zip` and `s2.dtx` with option `zap`.

The *options* are used to control which parts of the source files should be extracted to which generated file. A source file can contain a number of *modules*, and at the beginning of each module docstrip decides, for each output file separately, whether the code lines in that module should be extracted to the output file. The beginning of a module is marked by a guard line which has the syntax

`%<*`⟨*expression*⟩`>`

and the end by a corresponding

`%</`⟨*expression*⟩`>`

guard line. In their simplest form, the ⟨*expression*⟩s are names of options, and in that case the code lines in the module are only extracted if that option appears in the ⟨*options*⟩ for that combination of input file and output file. The ⟨*expression*⟩s can however be arbitrarily complicated boolean expressions; see [7] for more information. Modules may nest inside each other, and in that case the code lines in an inner module can only be included if all surrounding modules are being included. It is checked that matching `*` and `/` guard lines contain the same (as strings) ⟨*expression*⟩, and case is significant in the names of options.

One application of modules which has already been mentioned is to bundle code for several different generated files in the same `.dtx` file—one example of this is the file `doc.dtx` (part of the LaTeX base distribution) which contains both the doc package (`doc.sty`), the shortvrb package (`shortvrb.sty`), and two makeindex style files (`gglo.ist` and `gind.ist`). Another application is to keep variant sections of code—such as special code for debugging or gathering statistics—in the `.dtx` source file for a program without thereby making it a part of the normal form of that program. It is quite possible to use docstrip as a simple pre-processor for languages whose compiler/interpreter has not got one built in.

There are many other commands available in a docstrip installation script beside those listed above, but those are well described in the docstrip manual [7] and need little attention here. Instead I'm going to finish this subsection with a quick guide to the particular difficulties one faces when using docstrip to extract Tcl code, and how to overcome them.

The main problem is that docstrip insert a few comment lines at the beginning and end of each file it generates. This is a good thing, because a file consisting entirely of extracted code lines would normally be completely void of commentary and quite unintelligible for the casual user. These few comment lines explain that the file was generated by docstrip from other files (which contain the documentation of the code), lists those files, and normally also contains a copyright (or more commonly some kind of copyleft) notice. The problem lies in that comments look different in different languages, and as the default is to write TeX style comments, one must tell docstrip to write Tcl style comments. This can be done through the command

```
\edef\MetaPrefix{\string#}
```

which tells docstrip to begin each inserted comment line with the character '`#`'.

The comment lines inserted at the beginning of a generated file are called the *preamble* and those at the end the *postamble*. To set the preamble, one writes

`\preamble`
⟨*preamble lines*⟩
`\endpreamble`

and correspondingly to set the postamble

```
\postamble
⟨postamble lines⟩
\endpostamble
```

The ⟨*preamble lines*⟩ and ⟨*postamble lines*⟩ can be any number of lines (including zero). Unlike the text in source files, the text in these preamble and postamble lines is not read verbatim, so things in these lines which have special meaning to TEX (such as control sequences) will be treated as such; the only exception is that spaces and newlines are preserved (instead of concatenated to single spaces as they normally would). It is important that the preamble and postamble *are* set after `\MetaPrefix` is changed, because each line specified between `\preamble` and `\endpreamble` or `\postamble` and `\endpostamble` respectively will be prefixed by the current value of `\MetaPrefix`.

Finally, some programs (such as the UNIX core) assign special meaning to the first line of a file, so one might want to control what gets put there. Merely using `\preamble` doesn't achieve this, because the ⟨*preamble lines*⟩ specified that way are put after the lines saying "this is a generated file ...". You can however add things to the preamble by explicitly setting the macro `\defaultpreamble`, which is where docstrip stores the preamble. To make the first line a comment which simply contains the text '`-*-Tcl-*-`', you could give the command

```
\edef\defaultpreamble{\MetaPrefix\space -*-Tcl-*-^^J\defaultpreamble}
```

Similarly to begin the file by the three standard lines

```
#! /bin/sh
#\
exec tclsh "$0" ${1+"$@"}
```

(for an explanation see [9])—which on UNIX allow the file to function both as a Tcl script and a shell script which terminates the shell and runs `tclsh` on the script instead—you can use the command

```
\edef\defaultpreamble{%
   \MetaPrefix! /bin/sh^^J%
   \MetaPrefix\string\^^J%
   exec tclsh "$0" ${1+"$@"}^^J%
   \defaultpreamble
}
```

The full explanation of these commands is however far beyond this introduction.[2]

In summary, a docstrip installation script for extracting a file `foo.tcl` from `foo.dtx`, using Tcl style comments, inserting a BSD-style license notice in the preamble, and beginning with the line `# -*-Tcl-*-` could look as follows:

```
\input docstrip.tex

\edef\MetaPrefix{\string#}

\preamble
```

[2] Those who want to fully understand them should read *The TEXbook* [6], in particular Chapter 8.

```
Copyright (c) <YEAR>, <OWNER>
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
* Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following
disclaimer in the documentation and/or other materials provided
with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

\endpreamble

\postamble
\endpostamble

\edef\defaultpreamble{\MetaPrefix\space -*-Tcl-*-^^J\defaultpreamble}

\generate{\file{foo.tcl}{\from{foo.dtx}{bar}}}

\end
```

The generated file `foo.tcl` will contain

```
# -*-Tcl-*-
#
# This is file ‘foo.tcl’,
# generated with the docstrip utility.
#
# The original source files were:
#
# foo.dtx  (with options: ‘bar’)
#
# Copyright (c) <YEAR>, <OWNER>
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
```

```
# * Redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer.
# * Redistributions in binary form must reproduce the above
# copyright notice, this list of conditions and the following
# disclaimer in the documentation and/or other materials provided
# with the distribution.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
#
```

⟨*lines extracted from* `foo.dtx`⟩

```
#
#
# End of file 'foo.tcl'.
```

1.3 The structure of the LaTeX document

All this has been about the local appearance of a `.dtx` file, but what about the overall structure? There are several points to raise about that as well.

The first is that LaTeX to begin with treat `.dtx` documents just like any other document—a '`%`' starts a comment and only lines *not* beginning with a '`%`' contain anything that LaTeX can see. Somehow LaTeX must be instructed to start applying the special reading rules that were described above. This is the job of the so-called *driver*, which (for a file `myfile.dtx`) in its simplest form can look like

```
\documentclass{tclldoc}
\begin{document}
\DocInput{myfile.dtx}
\end{document}
```

The important command here is `\DocInput`, because that is what tells LaTeX to apply the special `.dtx` reading rules. More precisely it means "Start ignoring '`%`' characters in the text you read, input the file `myfile.dtx`, and when you're done return to treating '`%`' characters as before."

The driver is usually put in the very first stretch of code lines in the `.dtx` file. This means that LaTeX, when ordered to typeset the `.dtx` file, will start to read along, possibly ignoring hundreds of lines beginning with '`%`' because they are comments. Then it encounters the driver, and after the `\documentclass` and `\begin{document}` commands it executes the `\DocInput`. This will cause it to not ignore lines beginning with '`%`', so when it starts reading the file again it will see all the lines it skipped the first time through. The file will be read to end, after which LaTeX returns to the command after the `\DocInput`. As that command

happens to be `\end{document}`, it finishes the typeset document and stops. This stop prevents it from seeing and interpreting as LaTeX commands the remaining code lines in the file.

The second time through the driver shouldn't be interpreted as LaTeX commands, since for example the `\documentclass` command may only be used once in a LaTeX document. One way of achieving this is to put an `\iffalse` command right before the driver and a `\fi` command right after it. This says to LaTeX that the driver code is conditional material, and since the condition evaluates to false (`\iffalse` always evaluates to false), this conditional material should be skipped. Thus the first few lines of `myfile.dtx` typically might be

```
% \iffalse
%<*driver>
\documentclass{tclldoc}
\begin{document}
\DocInput{myfile.dtx}
\end{document}
%</driver>
% \fi
```

The `driver` guard lines are there to stop docstrip from including the driver code in the generated files.

After the driver comes the actual LaTeX document. The document usually consists of two parts, where the first part is a manual for the *usage* of the code defined in the file, and the second part contains the actual *implementation* (documented code). The idea is that most people are (at least the first time) quite content with learning how to use something, so one should make it simple for them to find that information.[3] To further this approach one puts the command `\StopEventually` at the start of the implementation part and the command `\Finale` at the end of it. Normally `\StopEventually` doesn't make itself felt, but if one previously has given the command `\OnlyDescription` then rest of the file will not be read; this can be used to produce a "manual only" version of the documentation. `\StopEventually` takes one argument and the code in this argument is executed at the `\Finale` (if the implementation part is being included) or immediately (if the implementation part isn't being included). Thus this argument is the place to put things that should appear at the very end of the document.

The `tcl`, `proc`, and `variable` environments described above all typically appear in the implementation part of the document.

2 Usage of commands and environments

2.1 The actual source code

`tcl` The `tcl` environment is used for wrapping up a group of code lines which contain Tcl code. Lines inside this environment which begin with a percent character are called *command* lines and can contain LaTeX commands which get executed, whereas lines that do not begin with a percent character are called *normal* lines and get typeset verbatim (or nearly verbatim). Lines that begin with `%<` (docstrip guard lines) do however constitute a special case, as the guard expression will get

[3] One needn't take this as an absolute rule—I for one haven't written all my packages that way—but structuring the document like this generally makes it more accessible.

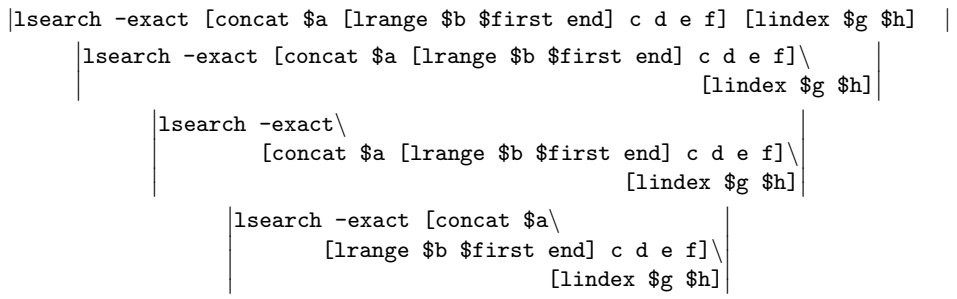

Figure 2: The same Tcl code, set in different linewidths

typeset as in doc's `macrocode` environment and the remainder of the line will get processed in command mode if it is a `*` or `/` guard, but in normal mode if the guard line was of any other type.

The `tcl` environment uses the same general formatting parameters as doc's `macrocode` environment. In particular this means that the text on a normal line is typeset in `\MacroFont` (by default the same thing as `\normalfont\ttfamily \small`) or `\AltMacroFont` (by default the same thing as `\normalfont\ttfamily \itshape\small`) depending on the current docstrip module nesting level. The tclldoc class sets the `StandardModuleDepth` counter to 1, which means that the `\AltMacroFont` is used when the modules are nested two levels deep or more.

If a normal line is too long to fit on one line then the `tcl` environment will try to break it. Legal breakpoints are spaces which could be replaced by a backslash and a newline without changing the meaning of the command; thus most spaces are legal breakpoints. When a line is broken at a space like this, the space is replaced by a backslash so that the line is still "syntactically correct". The opposite happens to lines which actually end with an escaped newline; such lines are concatenated with the following line and are treated as one long line. This is so that a sequence of breakpoints can be chosen which is optimal for the actual line width of the document (as opposed to the line width used in the text file, which can be something quite different). For example

```
% \begin{tcl}
lsearch -exact\
   [concat $a [lrange $b $first end] c d e f]\
   [lindex $g $h]
% \end{tcl}
```

could be typeset as any of the code examples in Figure 2, depending on how wide a line is. The environment tries to put the linebreaks at the lowest possible nesting (of braces and brackets) level; I believe this gives the best readability.[4] There is however a way to override this automatic choice of breakpoints: if a normal line which ends with an escaped newline is followed by a command line (even a command line that doesn't contain any commands) then it will not be concatenated with the next normal line.

[4]When I first started programming in Tcl I used a completely different method for breaking long lines in the code—I mainly implemented the current method because it was simple to program—but I was quite surprised by how readable it made the code.

There are a couple of restrictions on the code in command lines. First of all it is not allowed to start a new paragraph (there will be an error message). Secondly a command may not be broken across several lines—all the arguments must appear on the same line as the control sequence. Thirdly some characters have other catcodes than in normal LaTeX, so it is not certain that all commands work. Some commands that do work and may be useful are:

- Vertical space commands (`\smallskip`, `\medskip`, etc.) The command line '`% \medskip`' is more to type than a blank normal line, but it looks slightly better.
- Indexing commands (`\index`, `\IndexEntry`,[5] etc.)
- The `\TclInput` and `\settabsize` commands (see below).

And of course the `\end{tcl}` command works in a command line, since that is how one ends a `tcl` environment.

`tcl*` Besides the `tcl` environment there is also a `tcl*` environment which is different from `tcl` only in that spaces and tabs are typeset as special visible space '␣' and visible tab '⟶|' characters. This can be useful for pieces of code where the exact number of spaces in a sequence is significant, such as code for writing tables that align.

`\tclverb` For shorter pieces of Tcl code, e.g. examples, there's the `\tclverb` command. `\tclverb` is very similar to the standard LaTeX command `\verb`, but there are two differences. The first is that text typeset by `\tclverb` can contain breakpoints at whitespace; these behave just as in the `tcl` environment. The second is that the verbatim text that follows `\tclverb` may contain newlines, provided that these newlines are escaped by a backslash. Like a Tcl interpreter, `\tclverb` ignores whitespace following an escaped newline. Unlike a Tcl interpreter, `\tclverb` also ignores one percent character before the ignored whitespace, if it is the first character on the following line. Thus

```
% \tclverb|append a $b| is much more efficient than \tclverb|set a\
% $a$b| if \tclverb|$a| is long.
```

is perfectly legal, and the escaped newline between `a` and `$a$b` is treated just like the space between `set` and `a`. Like `\verb`, the `\tclverb` command has a starred `\tclverb*` form `\tclverb*` which also typesets spaces and tabs as visible characters.

`\MakeShortTclverb` The `\MakeShortTclverb` command works just like the doc/shortvrb command `\MakeShortVerb`, except that it makes the active character a shorthand reference for `\tclverb`... instead of `\verb`.... Use `\DeleteShortVerb` to undo the effect of a `\MakeShortTclverb`. The tclldoc class executes the command

```
\MakeShortTclverb{\|}
```

which makes `|` a shorthand for `\tclverb|`.

`\settabsize` Since there is no universally accepted standard for the size (equivalent number of spaces) of a tab, there is a command `\settabsize` for changing this. `\settabsize` takes as its only argument the new tab size, which must be an integer in the range 2–255. The default value is 8. `\settabsize` makes a local assignment to the tab size. The tab size can be changed inside a `tcl` (or `tcl*`) environment.

\TclInput

There is also a command `\TclInput` which is used for typesetting "raw" (not in `.dtx` format) Tcl code files. `\TclInput` is meant to be used on a *command* line of a `tcl` or `tcl*` environment, and it efficiently makes things look as if the `\TclInput` command had been replaced by the inputted file in its entirety (preceded by a newline, and followed by a percent and a space). `\TclInput` takes as its only argument the name of the file to input.

To typeset the file `myscript.tcl` one would write

```
% \begin{tcl}
% \TclInput{myscript.tcl}
% \end{tcl}
```

or even

```
\begin{tcl}\TclInput{myscript.tcl}\end{tcl}
```

anywhere in a tclldoc document. This works since the `tcl` environment is in command mode right after the initial `\begin{tcl}`, and the `\end{tcl}` needs not be the first command on a command mode line.

2.2 Markup of named things

proc
variable
arrayentry
arrayvar

The two environments `proc` and `variable`, which are analogues of `doc`'s `macro` environment, for procedures and variables respectively have already been mentioned in Section 1. In addition to those there are two environments `arrayentry` and `arrayvar` which are meant for entries in array variables and array variables as a whole. The complete syntaxes of these environments are

```
\begin{proc}[⟨namespace⟩]{⟨proc name⟩}
  ⋮
\end{proc}
\begin{variable}[⟨namespace⟩]{⟨variable name⟩}
  ⋮
\end{variable}
\begin{arrayentry}[⟨namespace⟩]{⟨array name⟩}{⟨entry name⟩}
  ⋮
\end{arrayentry}
\begin{arrayvar}[⟨namespace⟩]{⟨array name⟩}[⟨index-des⟩]
  ⋮
\end{arrayvar}
```

The ⟨*proc name*⟩, ⟨*variable name*⟩, and ⟨*array name*⟩ arguments are quite evidently the names of the procedure, variable, and array respectively. The ⟨*namespace*⟩ argument can be to specify the namespace part of a qualified name; having the name split like this makes it easier to treat the namespace differently from the rest of the qualified name. The command `\buildname` is used by the commands and environments described here to construct a qualified name from a namespace and a name. If there is no ⟨*namespace*⟩ argument then the namespace used will be the default namespace. The default namespace is set using the

\buildname

\setnamespace

\setnamespace command, which takes the namespace name as its only argument. The default namespace at the beginning of the document is the global namespace, whose name is the empty string.

The arrayentry environment is intended for certain distinguished entries in an array, such as entries inserted to make the boundary cases of an algorithm work correctly and entries which have a special meaning to the program. Not all arrays contain such special entries, but when they do it is a good practice to explain them explicitly. The ⟨*index-des*⟩ argument of the arrayvar environment can be used to specify what is used as index into the array; the text in this argument will appear both in the margin and in the index, but note that ⟨*index-des*⟩ is a moving argument. There is little difference between the variable and arrayvar environments when the ⟨*index-des*⟩ argument of the latter isn't used, but the index entries they make behave differently with respect to arrayentry index entries. An arrayentry index entry will be put as a subentry of the corresponding arrayvar entry, whereas a variable entry would appear separately.

\describestring

The above environments usually only appear in the implementation part of a .dtx file. For the usage part there is a command \describestring which produces marginal headings and index entries. The syntax of \describestring is

\describestring[⟨*type*⟩][⟨*namespace*⟩]{⟨*text*⟩}

The ⟨*text*⟩ is the string for which a heading and index entry will be made, whereas the ⟨*type*⟩ (if given) is put after the text. If the ⟨*namespace*⟩ is given then the thing described is supposed to be the name of something namespace-relative (like a procedure or global variable) and in this case the complete name is formed by passing ⟨*namespace*⟩ and ⟨*text*⟩ to \buildname. If ⟨*type*⟩ is proc, var., or array and a namespace is given then the index entry made will fit that made by a corresponding proc, variable, or arrayvar respectively environment. The ⟨*type*⟩ argument is, in LaTeX terminology, moving.

The ⟨*text*⟩ and ⟨*namespace*⟩ arguments can contain arbitrary characters and most characters can be entered verbatim. Amongst the exceptions are '%', '\', '{', and '}', which instead can be entered as \PrintChar{`\%}, \PrintChar{`\\}, \PrintChar{`\{}, and \PrintChar{`\}} respectively. See the xdoc package [4] documentation for an explanation of the \PrintChar command. The ⟨*text*⟩ argument can also contain "variant" parts made using the \meta command. As an example,

\meta

\describestring[array]{\meta{mode}modeVars}

puts the text

⟨*mode*⟩modeVars (array)

in the margin and index. The arguments of such \meta commands are moving.

\describeopt

A case which deserves special treatment is that of options of commands and for that there is the \describeopt command. The syntax of this command is

\describeopt*[⟨*namespace*⟩]{⟨*command*⟩}[⟨*type*⟩]{⟨*option*⟩}

[5] This command is defined by the xdoc package [4].

where ⟨*command*⟩ is the command whose option is being described and ⟨*option*⟩ is the name of the option. ⟨*namespace*⟩ is the namespace of the ⟨*command*⟩ and defaults to the global namespace. ⟨*type*⟩ is the type of the command and defaults to `proc` for procedure. The ⟨*namespace*⟩, ⟨*command*⟩, and ⟨*type*⟩ are (currently) only used in the index entry that is generated. Options for procedures will be put as subentries of the main procedure entry. For built-in commands it might be more appropriate to use `command` as ⟨*type*⟩, e.g.

```
\describeopt{lsort}[command]{-real}
```

will put "`-real` option" in the margin and in the index as a subentry of "`lsort` (command), global namespace". The `*` is optional—including it will supress the marginal note normally generated by this command.

`Cfunction` `Cvariable` `Ctype`

Since Tcl is often used together with C, it might be useful to also have something similar to the `proc` environment for C things. This is what the `Cfunction`, `Cvariable`, and `Ctype` environments are for. These take as their only argument the name of the identifier that is defined, e.g.

```
\begin{Cfunction}{main}
```

`\Cfunctionidentifier` `\Cvariableidentifier` `\Ctypeidentifier`

Since there seems to be several schools on how C code should be formatted when typeset, the formatting of identifiers passed to these environments is configurable. The three commands `\Cfunctionidentifier`, `\Cvariableidentifier`, and `\Ctypeidentifier` handle all typesetting of identifiers; each takes as its only argument the identifier (as a harmless string) to typeset. The default is to set the argument in italic; this is what CWEB does.

`notrawchar` option

If you are using the `C...` environments for identifiers whose names contain underscores (_), you may want to pass the `notrawchar` to `tclldoc` (it is really an option of the `xdoc` package and will be passed on to that automatically). This option addresses a problem with `OT1`-encoded fonts that may cause underscores to display as a quite different character (the `cmtt` typewriter fonts are however not affected by this problem).

2.3 Describing command syntaxes

One important part of documentation is to describe the syntaxes of commands. The previous subsection contains examples of the conventions for this that has been developed for LaTeX commands—mandatory arguments are denoted as '`{`⟨*argument*⟩`}`' and optional arguments are denoted as '`[`⟨*argument*⟩`]`'. These two classes suffice rather well for LaTeX commands, but the syntaxes of Tcl commands are not seldom much more complex. Therefore a more powerful form of syntax specification is called for, and one which is close at hand is that used in regular expressions since it is already part of the Tcl language anyway.

`\regopt` `\regstar` `\regplus`

The simplest commands available are the modifiers `\regopt`, `\regstar`, and `\regplus`, which correspond to the `?`, `*`, and `+` metacharacters in a regular expression; using `\regopt` after a term says that it is optional, `\regstar` says that the term can be repeated an arbitrary number of times (including zero), and `\regplus` says that the term occurs at least once. The typeset results of these commands are $^{?}$, *, and $^{+}$ respectively (recall that exponents are sometimes used to denote repetition).

`\word`

The terminals in the expression are best made using `\tclverb` (for "fixed" material, e.g. procedure names) and `\word` (for variable material, e.g. arguments).

The syntax of `\word` is

`\word{`⟨*text*⟩`}`

and e.g. `\word{script}` gets typeset as

{*script*}

Using these, one can for example specify the syntaxes of the Tcl commands `append` and `catch` through

```
|append| \word{var-name} \word{value}\regplus
|catch| \word{script} \word{var-name}\regopt
```

(recall that '`|`' is a shorthand for '`\tclverb|`'). These get typeset as

`append` {*var-name*} {*value*}$^{+}$
`catch` {*script*} {*var-name*}$^{?}$

`regblock`

Terms in regular expressions can also consist of parenthesised subexpressions, which are made using the `regblock` environment. The syntax of this environment is

`\begin{regblock}[`⟨*modifier*⟩`]` ... `\end{regblock}`

If `regblock` environments are nested then the parentheses of the outer environment will grow to be larger than those of the inner environment. A side-effect of this is that the `regblock` environment wants to know if a modifier will be applied to it, since the amount by which the modifier should be raised in this case depends on the size of the parenthesis before it, and this is what the ⟨*modifier*⟩ optional argument is for. LaTeX does not provide for arguments at the `\end` of an environment, so it has to be placed at the `\begin`. Using these elements, the syntax of `if` can be specified through

```
|if| \word{expression} |then|\regopt\ \word{script}
\begin{regblock}[\regstar]|elseif| \word{expression}
|then|\regopt\ \word{script}\end{regblock}
\begin{regblock}[\regopt]|else| \word{script}\end{regblock}
```

which typesets as

`if` {*expression*} `then`$^{?}$ {*script*} (`elseif` {*expression*} `then`$^{?}$ {*script*})* (`else` {*script*})$^{?}$

`regexp`

In versions of tclldoc before 2.40, the `regblock` environment used to be called `regexp`. That other name is still supported, but should be avoided in new documents.

`\regalt`

The final regular expression construction that is supported is that of branches of a regular expression. A `regblock` environment consist of one or several branches that are separated by `\regalt` commands. Visually the `\regalt` command gets typeset as a vertical bar that has the same size as the parentheses of the surrounding `regblock` environment. The `\regalt` command may only be used inside a `regblock` environment. An example of the use of `\regalt` is the following specification of the syntax of Tcl's `regexp` command:

```
|regexp| \begin{regblock}[\regstar]|-nocase|\regalt
|-indices|\end{regblock} |--|\regopt \word{regular expression}
\word{string} \word{var-name}\regstar
```

which typesets as

> `regexp` (`-nocase` | `-indices`)* `--`? {*regular expression*} {*string*} {*var-name*}*

Finally a note about the relationship between the `\word` command and `doc`'s `\meta` command. Whereas the argument of `\word` is encapsulated in braces (and thus ought to be a separate word for a Tcl interpreter), the argument of `\meta` is encapsulated in angle brackets. The idea is that `\word` should be used for things which are separate words to Tcl, whereas `\meta` should be used for things which corresponds to parts of words or to several words. Thus in the command `set b Z${a}Y`, the second word `b` would be a '{*var-name*}' and the third word `Z${a}Y` would be a '`Z`⟨*string*⟩`Y`'. In the command `label .a -text "Hello there!"`, the last two arguments could be summarised as an ⟨*option*⟩, but not as an {*option*}.

2.4 Non-ASCII characters

One problem, which is only going to be more common in the future, is how to deal with non-ASCII characters in scripts. The main problem here lies not on the output side, as LaTeX is actually pretty good at producing a requested character as long as it is available in some font, but on the input side. LaTeX can handle input in most 8-bit encodings, but in order for that to work the file must contain an `\inputencoding` command which tells LaTeX which encoding is being used. As transporting a file from one platform to another most likely changes the encoding, but not the argument of `\inputencoding`, this method is rather fragile. Certainly there is room for improvements but the world of 8-bit encodings is generally such a mess anyway that it probably isn't worth the effort.

A more progressive approach is to decide that all source code is in Unicode (more precisely in UTF-8). The main arguments for this are: (i) Tcl uses Unicode internally, (ii) it is equally foreign on all platforms and can be treated as binary data rather than "extended ASCII" text, and (iii) since it isn't converted, there is no loss of data. Interestingly enough, *it is possible to use UTF-8 "out of the box" today!* Using the `ucs` package [10] allows LaTeX to interpret UTF-8 input and this works just as well for the Tcl code in a `tcl` environment as for the normal LaTeX text outside it. If `docstrip` is run on a LaTeX format[6] that preserves characters whose most significant bit is set[7] then the non-ASCII characters are simply copied verbatim and it makes no difference that they may occupy more than one byte of data. Alternatively one can run `docstrip` on Omega and (with a little extra work) get the ability to have the Tcl code translated to some other encoding as the files are being generated![8]

But although the above paragraph describes the way to go in the long run, there are some matters which make this approach slightly unfeasible in the near future. This is of course my own subjective opinion, but I find that two good

[6] Or, in some implementations, the TeX program gets a suitable option.
[7] Rather than converting them to `^^`-sequences, which is the default.
[8] At least in theory; I have to admit I haven't actually tested the `docstrip` part of it.

reasons not to start using Unicode throughout quite yet are that (i) my favourite text editor doesn't support Unicode (yet) and (ii) even if I do start using it, there wouldn't be that much people around who could make sense of such files if I were to send it to them. Therefore I *intend* to implement, but as yet haven't, a kind of intermediate format where non-ASCII Unicode characters are encoded using only ASCII characters plus an extra escape character. The basic idea is simply that any string '⟨*escape*⟩⟨*hex digits*⟩⟨*escape*⟩' should be interpreted as the Unicode character `U+`⟨*hex digits*⟩, so that arbitrary Unicode characters can be encoded using a character set that only comprises ASCII plus one extra ⟨*escape*⟩ character. Supposing that this ⟨*escape*⟩ character is the centered dot '·', I could then encode my name as

```
Lars Hellstr·00f6·m
```

whereas the centered dot itself would be `·00b7·`. The idea is that the file itself should contain the declaration of which character is used as this Unicode escape, so that a change due to translation from one 8-bit encoding to another will identically alter both declaration and use of the escape character, thereby preserving the internal logic of the file.

The weak point with this scheme is that docstrip would have to translate the escape sequences to proper characters when it generates files. Implementing that under TEX is highly non-trivial. It can be done with a reasonable effort under Omega, but it still requires hacking docstrip. The really interesting approach would however be to implement it in a port of docstrip to Tcl, as that would remove the need to have TEX to install the files. Porting docstrip to Tcl is by the way a project of mine which I unfortunately haven't spent much time on, but if it is to be of any use to have the Unicode escape format described above implemented in tclldoc then I will have to make some progress with it.

One rather recent advancement in this direction is the code in `sourcedtx.tcl`, which can be generated from the file `sourcedtx.dtx` that is distributed with tclldoc as an example. This implements a Tcl command `dtx::source` that makes it possible to source Tcl code in a `.dtx` file without docstripping it to a file first. This code does currently not bother about encodings, but that is easy enough to add.

Finally, a few notes on the old mechanism for non-ASCII characters that is included in the tcldoc compatibility class. It cooperates with the rtkinenc package [3], when that has been loaded, in order to detect when an input character isn't avaiable in any active font encoding. Rather than raising an error and printing nothing it these cases, missing characters are written as the corresponding `\x`⟨*hh*⟩ backslash sequence in a slightly different font than the rest of the text. A problem here is however that most input encodings contain a few characters which are interpreted as *math* character by LATEX. When such a character appears on a code line it makes TEX switch to math mode and things generally get quite messy afterwards.

The cure for this is to redeclare these input characters to LATEX so that they work as intended in text mode, but that does take some lines of code. The tcldoc class does contain the declarations needed for the `applemac` input encoding; passing it the `macinputenc` option will load the rtkinenc package, the `applemac` (macRoman) input encoding, the `TS1` output encoding, and make the necessary redeclarations to allow all input characters to work in text. As nothing is provided

`macinputenc` option

for any other input encoding however, this solution never was a good solution. The `macinputenc` should be considered as unsupported as of tclldoc v 2.30.

2.5 Options and customisation

The tclldoc package does not have any options of its own, but all options passed to it are passed on to the xdoc package. The tclldoc class accepts all options that the standard LaTeX document class article accepts, with the exception of `a5paper`.

Like the ltxdoc class, the tclldoc class will look for a special configuration file `tclldoc.cfg` and input that file if it exists. This file can be used to declare extra options for the class, have certain options always given, etc. Section 2 of `ltxdoc.dtx` [2] is a good introduction to how such configuration files can be used with `.dtx` sources in general.

When you use a `tclldoc.cfg` file to customise the tclldoc document class, you affect how all documents using that class will be typeset in your particular TeX installation. It is *not* something you have to do, but it can make tclldoc documents work better with the printers, paper formats, fonts, etc. that are available in your installation. It will usually cause line and page breaks to occur at other places than they would do if typeset using an uncustomised tclldoc class, so the typographical quality of the document can be decreased, but it is uncommon to find an `.dtx` document whose author have given these matters much attention anyway. Hence the typographic arguments against customisation are weak.

A common form of customisation is to use additional packages, since various kinds of document-wide font selection is often done by packages. Due to that the code in `tclldoc.cfg` is executed when the tclldoc class does its option processing, at which time LaTeX does not allow loading packages, such customisation is not straightforward. There is a way around that however; to load e.g. the times package, use the command

```
\AtEndOfClass{\usepackage{times}}
```

Using `\AtEndOfClass` like this delays the command until it may be executed.

2.6 Miscellanea

`\Tcllogo` For writing "Tcl", the tclldoc package defines the command `\Tcllogo`, which for most fonts look slightly better than simply typing `Tcl`. (`\Tcllogo` becomes Tcl, whereas `Tcl` becomes Tcl.)

Between the namespace and the tail part of a qualified name, the tclldoc package commands naturally put the namespace separator '`::`'. This text is stored
`\namespaceseparator` in the macro `\namespaceseparator`, which can be redefined using the `\DeclareRobustCommand` command. This is mainly useful for modifying how this separator behaves with respect to line breaking; the default behaviour is that a line break can occur between the colons.

`\namespacephrase` Another configurable piece of text is stored in the `\namespacephrase` macro. This contains word 'namespace' as that appears in index entries, e.g. in the last word of

> `platform` (var.), `alpha` namespace

It is often convenient to replace this by something shorter. The redefinition

```
\renewcommand{\namespacephrase}{\textsc{ns}}
```

turns the above into

`platform` (var.), `alpha` NS

Note however that either `\namespacephrase` itself or its expansion must be robust.

3 Acknowledgements

The tclldoc document class and LaTeX package were constructed starting from three other sources: (i) the ltxdoc document class [2] by David Carlisle, (ii) the doc package [8] by Frank Mittelbach, B. Hamilton Kelly, Andrew Mills, Dave Love, and Joachim Schrod, and (iii) my own pasdoc document class. Hence the 'et al.' in the author field above. This complicated heritage in the code is mirrored by the documented source—there are paragraphs below that are rather about one of (i)–(iii), than about tclldoc.

References

[1] Johannes Braams, David Carlisle, Alan Jeffrey, Frank Mittelbach, Chris Rowley, and Rainer Schöpf: *ltoutenc.dtx*, The LaTeX3 Project; CTAN:`macros/latex/base/ltoutenc.dtx`.

[2] David Carlisle: *The file `ltxdoc.dtx` for use with LaTeX 2ε*, The LaTeX3 Project; CTAN:`macros/latex/base/ltxdoc.dtx`.

[3] Lars Hellström: *The rtkinenc package*, 2000; CTAN:`macros/latex/contrib/supported/rtkinenc/rtkinenc.dtx`.

[4] Lars Hellström: *The xdoc package — experimental reimplementations of features from doc, second prototype*, 2000, 2001; CTAN:`macros/latex/exptl/xdoc/xdoc2.dtx`.

[5] Lars Hellström: *The docindex package*, 2001; CTAN:`macros/latex/exptl/xdoc/docindex.dtx`.

[6] Donald E. Knuth, Duane Bibby (illustrations): *The TeXbook*, Addison–Wesley, 1984; ISBN 0-201-13448-9 and 0-201-13447-0.

[7] Frank Mittelbach, Denys Duchier, Johannes Braams, Marcin Woliński, and Mark Wooding: *The DocStrip program*, The LaTeX3 Project; CTAN:`macros/latex/base/docstrip.dtx`.

[8] Frank Mittelbach, B. Hamilton Kelly, Andrew Mills, Dave Love, and Joachim Schrod: *The doc and shortvrb Packages*, The LaTeX3 Project; CTAN:`macros/latex/base/doc.dtx`.

[9] "Tom", Donal Fellows, Larry Virden, Richard Suchenwirth: *exec magic*, The Tcl'ers Wiki page **812**; HTTP://`mini.net/tcl/812.html`.

[10] Dominique P. G. Unruh: *`ucs.sty` - Unicode Support*, 2000; CTAN:`macros/latex/contrib/supported/unicode/`.

The "CTAN:" above is short for "any of the servers in the Comprehensive TEX Archive Network (or mirror thereof)". You get a working URL if you replace this by e.g. "`ftp://ftp.tex.ac.uk/tex-archive/`".