LaTeX

Good things come in little packages: An introduction to writing .ins and .dtx files

Scott Pakin

Abstract

LaTeX packages made available from CTAN are commonly distributed as a pair of files: ⟨*something*⟩.ins and ⟨*something*⟩.dtx. The user is then instructed to run the .ins file through latex to produce the actual package files. What are these .ins and .dtx files? How do you, as a class or style-file writer, create your own? And, why would you want to? This article answers those questions and elucidates the mysterious techniques underlying LaTeX package distribution.

1 Introduction

A typical CTAN package comprises a README file, some PDF documentation, an .ins file, and a .dtx file. Running the .ins file through latex creates one or more .sty, .cls, .def, or other files that the user can install. Few LaTeX users and developers understand the reasoning behind that extra step or the purpose of the seemingly unnecessary .dtx file.

Before we examine .ins and .dtx files in depth, let us consider a coding example from outside the TeX world. Figure 1 presents a function in the C programming language for solving a quadratic equation. Comments at the top of the function are used to explain what the function does. Although comments are intended to be human-readable, only simple text can be used to format comments. Wouldn't it be nice if comments and the code they describe could be typeset using a tool such as LaTeX, as in Figure 2? Even for short programs, including mathematics, figures, and tables in comments can assist readability. For longer programs, sectioning commands, cross references (maybe even with hyperlinks), indexes, and a table of contents can be quite beneficial for explaining the program's purpose and usage to readers. However, having to maintain two versions of a program — a nicely formatted version with typeset documentation for human readers and a text-only version for the compiler — is an approach doomed to failure as the two versions will inevitably drift apart.

The idea behind a .dtx file is to maintain a single version of a program yet be able to process it either as a typeset document or as compilable code. As far as the latex compiler is concerned, a .dtx file is an ordinary document; it just happpens to describe a program. However, when the corresponding .ins file is processed, the (text-only) program is extracted from the .dtx file to one or more separate files.

Placing emphasis on providing thorough, typeset code documentation intertwined with the code itself is commonly known as *literate programming* [1]. Literate programming is particularly apropos for documenting LaTeX packages because of the esotericism of the TeX language and the consequent need for copious explanation. The mechanisms needed to implement literate programming in .ins and .dtx files are provided by two packages that come standard with all LaTeX 2_ε distributions: Doc [3, 4] for typesetting, formatting, and indexing LaTeX macro and environment definitions, and, DocStrip [5] for extracting the code from a literate program while stripping away all of the commentary.

2 Installer (.ins) files

The first step in preparing a package for distribution is to write an *installer* (.ins) file. An installer file extracts the code from a .dtx file, uses DocStrip to strip off the comments and documentation, and outputs a .sty file. The good news is that a .ins file is typically fairly short and doesn't change significantly from one package to another.

Figure 3 presents a typical .ins file. An .ins file usually begins with a comment block that states the package's copyright notice and license agreement (lines 1–13). Most of the commands that appear in an .ins file are provided by the DocStrip package so that is loaded in line 15. DocStrip is normally excessively verbose about its operation so Figure 3 includes an invocation of \keepsilent (line 16) to instruct DocStrip to output only the most important information.

A package can invoke the \usedir macro (as on line 18) to specify a preferred installation directory relative to the root of the TeX directory tree. In practice, the \usedir call serves primarily as a comment and is seldom used to automatically place files in their final destination.

Lines 20–36 of Figure 3 specify a set of comments to include at the top of every file that the .ins file generates. Typically, these comments include a remark that the file is generated plus a repetition of the package's copyright notice and license agreement.

The most important line in an .ins file is the call to \generate. The \generate macro is the mechanism by which an .ins file instructs DocStrip how to extract the various package files from an accompanying .dtx file. Line 38 of Figure 3 should be interpreted as the instruction, "Generate a file called mypackage.sty by extracting all text marked with

```
/* Use the quadratic formula (x=(-b +/- sqrt(b^2-4ac))/2a) to store the two
 * roots of ax^2+bx+c=0 in x1 and x2.  Return 1 on success, 0 on failure
 * (if a=0 or the roots are complex). */
int solve_quadratic (double a, double b, double c, double *x1, double *x2)
{
  double discrim = b*b - 4*a*c;

  if (a == 0.0 || discrim < 0.0)
    return 0;
  *x1 = (-b + sqrt(discrim)) / (2*a);
  *x2 = (-b - sqrt(discrim)) / (2*a);
  return 1;
}
```

Figure 1: Compiler-readable C code for solving a quadratic equation

solve_quadratic() Use the quadratic formula ($x = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$) to store the two roots of $ax^2+bx+c=0$ in x_1 and x_2. Return `1` on success, `0` on failure (if $a = 0$ or the roots are complex).

```
int solve_quadratic (double a, double b, double c, double *x1, double *x2)
{
    double discrim = b*b − 4*a*c;

    if (a == 0.0 || discrim < 0.0)
      return 0;
    *x1 = (−b + sqrt(discrim)) / (2*a);
    *x2 = (−b − sqrt(discrim)) / (2*a);
    return 1;
}
```

Figure 2: Typeset C code for solving a quadratic equation

the tag `package` in the file called `mypackage.dtx`."

The `\generate` command is fairly flexible in that a file can be generated from multiple tags spread across multiple files. In fact, blocks of code can be shared by multiple generated files. As a fairly complex example, consider the `\generate` command used by LaTeX 2ε's `classes.ins` file to generate all of the standard LaTeX 2ε class files and their per-size helper files. As the excerpt from `classes.ins` shown in Figure 4 indicates, both `size10.clo` and `bk10.clo` are produced by extracting all text from `classes.dtx` that is marked with the `10pt` tag. The `bk10.clo` file additionally includes all text marked with the `bk` tag. The same `bk`-tagged text is copied into `bk11.clo` and `bk12.clo` as well.

Returning to our complete example of an `.ins` file in Figure 3, DocStrip provides a `\Msg` macro that outputs a message to the standard output device. It is helpful to use `\Msg` to inform the user what files were extracted and need to be installed. Lines 40–53 in Figure 3 output a typical end-of-installation message. Note the use of `\obeyspaces` in line 40 to prevent TeX from collapsing multiple spaces into a single space in the subsequent `\Msg` invocations.

An `.ins` file ends with a call to `\endbatchfile`, as shown in line 55.

3 Documented LaTeX (`.dtx`) files

A *documented LaTeX* (`.dtx`) file contains both the commented source code and the user documentation for the package. Running a `.dtx` file through `latex` typesets the user documentation, which usually also includes a nicely typeset version of the commented source code.

Due to some Doc trickery, `latex` actually evaluates a `.dtx` file *twice* when generating documentation. On the first pass, only a small piece of `latex` driver code is evaluated. The second time, *comments* in the `.dtx` file are evaluated, as if there were no "`%`" preceding them. This can lead to a great deal of confusion when writing `.dtx` files and occasionally leads to some awkward constructions. Fortunately, once the basic structure of a `.dtx` file is in place, filling in the code is fairly straightforward.

```
%%
%% Copyright (C) 2008 by Your Name Here <you@yournamehere.org>
%%
%% This file may be distributed and/or modified under the conditions of
%% the LaTeX Project Public License, either version 1.3c of this license
%% or (at your option) any later version.  The latest version of this
%% license is in:
%%
%%    http://www.latex-project.org/lppl.txt
%%
%% and version 1.3c or later is part of all distributions of LaTeX
%% version 2006/05/20 or later.
%%

\input docstrip.tex
\keepsilent

\usedir{tex/latex/mypackage}

\preamble

This is a generated file.

Copyright (C) 2008 by Your Name Here <you@yournamehere.org>

This file may be distributed and/or modified under the conditions of
the LaTeX Project Public License, either version 1.3c of this license
or (at your option) any later version.  The latest version of this
license is in:

   http://www.latex-project.org/lppl.txt

and version 1.3c or later is part of all distributions of LaTeX
version 2006/05/20 or later.

\endpreamble

\generate{\file{mypackage.sty}{\from{mypackage.dtx}{package}}}

\obeyspaces
\Msg{*************************************************************}
\Msg{*                                                           *}
\Msg{* To finish the installation you have to move the following *}
\Msg{* file into a directory searched by TeX:                    *}
\Msg{*                                                           *}
\Msg{*    mypackage.sty                                          *}
\Msg{*                                                           *}
\Msg{* To produce the documentation run the file mypackage.dtx   *}
\Msg{* through LaTeX.                                            *}
\Msg{*                                                           *}
\Msg{* Happy TeXing!                                             *}
\Msg{*                                                           *}
\Msg{*************************************************************}

\endbatchfile
```

Figure 3: A typical `.ins` file

```
\generate{\file{article.cls}{\from{classes.dtx}{article}}
          \file{report.cls}{\from{classes.dtx}{report}}
          \file{book.cls}{\from{classes.dtx}{book}}
          \file{size10.clo}{\from{classes.dtx}{10pt}}
          \file{size11.clo}{\from{classes.dtx}{11pt}}
          \file{size12.clo}{\from{classes.dtx}{12pt}}
          \file{bk10.clo}{\from{classes.dtx}{10pt,bk}}
          \file{bk11.clo}{\from{classes.dtx}{11pt,bk}}
          \file{bk12.clo}{\from{classes.dtx}{12pt,bk}}
          }
```

Figure 4: Excerpt from $\LaTeX\,2_\varepsilon$'s `classes.ins` file

```
1  % \iffalse meta-comment
2  %
3  % Copyright (C) 2008 by Your Name Here <you@yournamehere.org>
4  % -----------------------------------------------------------
5  %
6  % This file may be distributed and/or modified under the conditions of
7  % the LaTeX Project Public License, either version 1.3c of this license
8  % or (at your option) any later version.  The latest version of this
9  % license is in:
10 %
11 %    http://www.latex-project.org/lppl.txt
12 %
13 % and version 1.3c or later is part of all distributions of LaTeX
14 % version 2006/05/20 or later.
15 %
16 % \fi
17 %
```

Figure 5: `.dtx` header comments

3.1 Package identification

A `.dtx` file traditionally begins with a copyright and license notice, which are formatted as in Figure 5. The significance of the `\iffalse`...`\fi` construct is that on `latex`'s second pass through the `.dtx` file, commented lines are processed as if they were uncommented. To prevent the copyright and license statement from appearing at the beginning of the typeset document we wrap them within a conditional that will never be true. The `meta-comment` after `\iffalse` is nothing more than a convention for indicating that the comment is intended to be read by a human, not by Doc, DocStrip, or `latex`.

The next block of `.dtx` code (Figure 6) identifies the package. On `latex`'s first pass through the `.dtx` file, "`%`" introduces a comment line, as normal. Hence, `latex` sees only the `\ProvidesFile` command (line 20) and its optional argument (line 25). The optional argument must be in the format shown: package date (YYYY/MM/DD), package version, and package description. The Doc package parses the optional argument into three macros — `\filedate`, `\fileversion`, and `\fileinfo` — that can be used to automatically date-stamp and version-stamp the documentation. On `latex`'s second pass through the file, the `\iffalse`, which is now executed, tells `latex` to disregard the entire block of code shown in Figure 6.

```
18 % \iffalse
19 %<*driver>
20 \ProvidesFile{mypackage.dtx}
21 %</driver>
22 %<package>\NeedsTeXFormat{LaTeX2e}[2003/12/01]
23 %<package>\ProvidesPackage{mypackage}
24 %<*package>
25     [2008/02/18 v1.0 My sample package]
26 %</package>
27 %
```

Figure 6: `.dtx` package identification

The remaining lines of Figure 6 are ignored on both the first and second pass through the file. However, they still have an important purpose. In addition to the two `latex` passes over the `.dtx` file for producing documentation, `latex` is also run on the `.ins` file to extract the various package files from the `.dtx` file. The `\generate` call on line 38 of Figure 3 associated the tag `package` with the derived file `mypackage.sty`. Consequently, all lines either beginning with `%<package>` or bracketed between `%<*package>` and `%</package>` are written to `mypackage.sty`. Thus, the code in Figure 6 writes to `mypackage.sty` the `\NeedsTeXFormat` line (line 22), the `\ProvidesPackage` line (line 23), and the optional argument to `\ProvidesPackage` (line 25) — which, as we saw, cleverly also serves as the optional argument to `\ProvidesFile` when generating the package documentation.

`\NeedsTeXFormat` and `\ProvidesPackage` are part of the standard LaTeX 2_ε package-identification mechanism [6]. (Classes use `\ProvidesClass` instead of `\ProvidesPackage`, while other file types use `\ProvidesFile`.) `\NeedsTeXFormat` specifies the earliest date of the LaTeX format itself with which the package is compatible. (From LaTeX, `\show\fmtversion` displays the current format date.) The argument to `\ProvidesPackage` is written to the `.log` file associated with any document that uses the corresponding package.

3.2 Driver code

When producing documentation from a `.dtx` file, the *driver code* is the first block of code that `latex` sees. Figure 7 lists typical driver code. Because `mypackage.ins` does not supply a `\generate` rule for `driver`, placing the driver between `%<*driver>` and `%</driver>` ensures that it will not be processed when generating package files from the `.ins` file. `ltxdoc` is a class designed for typesetting LaTeX documentation; it derives from article but additionally includes the Doc package and defines a few useful commands for documenting classes and packages. One of those commands, `\EnableCrossrefs` (line 30), specifies that the document's index should automatically cross-reference the use of every control sequence (macro or primitive) in the package code. `\CodelineIndex` (line 31) indicates that references to code in the index should point to the corresponding line number instead of to the corresponding page number. `\RecordChanges` (line 32) says to create a file of package changes that can then be incorporated automatically into the documentation in a "Change History" section.

Within the document's body, the `\DocInput` call (line 34 of Figure 7) is the critical line. `\DocInput` tells the Doc package to input the `.dtx` file from within itself. In this second pass through the `.dtx` file, percent characters are not treated as comment

```
28 %<*driver>
29 \documentclass{ltxdoc}
30 \EnableCrossrefs
31 \CodelineIndex
32 \RecordChanges
33 \begin{document}
34   \DocInput{mypackage.dtx}
35   \PrintChanges
36   \PrintIndex
37 \end{document}
38 %</driver>
39 % \fi
```

Figure 7: The `.dtx` driver code

characters but are instead ignored. (The sequence "`^^A`" can be used instead of "`%`" to introduce a comment.) After the code is typeset, the `\PrintChanges` call (line 35) typesets a Change History section that informs the reader about the changes that were made to the source code in each revision. `\PrintIndex` (line 36) typesets an index. Finally, the `\fi` in line 39 matches the `\iffalse` in line 18 of Figure 6.

3.3 Code verification

The remainder of this section discusses the part of the `.dtx` file that is processed recursively by `\DocInput`: the documentation proper. In this part of the document, lines beginning with a percent sign are treated as documentation (i.e., the "`%`" is stripped and the result is processed as ordinary LaTeX code). Lines not beginning with a percent sign are both processed as documentation and written to the `.sty` file. This rigmarole is the key to using the same code in both a typeset document and a LaTeX package.

The documentation traditionally begins with a block of code that may be considered slightly anachronistic: a document checksum and a test for unexpected variations in character encoding. The `\CheckSum` call in line 40 of Figure 8 takes an argument representing the total number of backslash characters in the package code (i.e., in lines not beginning with a percent sign). If the tally is correct, Doc outputs

```
*******************
* Checksum passed *
*******************
```

If the tally is incorrect, Doc issues an error message:

`! Package doc Error: Checksum not passed (`⟨*incorrect*⟩`<>`⟨*correct*⟩`).`

If the tally is `0`, Doc outputs the correct tally but does not issue an error message:

```
40 % \CheckSum{0}
41 %
42 % \CharacterTable
43 %  {Upper-case    \A\B\C\D\E\F\G\H\I\J\K\L\M\N\O\P\Q\R\S\T\U\V\W\X\Y\Z
44 %   Lower-case    \a\b\c\d\e\f\g\h\i\j\k\l\m\n\o\p\q\r\s\t\u\v\w\x\y\z
45 %   Digits        \0\1\2\3\4\5\6\7\8\9
46 %   Exclamation   \!     Double quote  \"     Hash (number) \#
47 %   Dollar        \$     Percent       \%     Ampersand     \&
48 %   Acute accent  \'     Left paren    \(     Right paren   \)
49 %   Asterisk      \*     Plus          \+     Comma         \,
50 %   Minus         \-     Point         \.     Solidus       \/
51 %   Colon         \:     Semicolon     \;     Less than     \<
52 %   Equals        \=     Greater than  \>     Question mark \?
53 %   Commercial at \@     Left bracket  \[     Backslash     \\
54 %   Right bracket \]     Circumflex    \^     Underscore    \_
55 %   Grave accent  \`     Left brace    \{     Vertical bar  \|
56 %   Right brace   \}     Tilde         \~}
```

Figure 8: `.dtx` verification code

```
**********************************
* This macro file has no checksum!
* The checksum should be ⟨number⟩!
**********************************
```

It is convenient to specify `\CheckSum{0}` when developing a package and to replace `0` with the correct checksum only when the package is ready to be released.

The character table must appear exactly as shown in Figure 8, lines 42–56. Doc verifies that the character table has not been corrupted and outputs the following success message:

```
***************************
* Character table correct *
***************************
```

If any character differs from that which was expected, Doc issues the following error message:

```
! Package doc Error: Character table
corrupted.
```

3.4 Miscellaneous initialization

Doc can automatically typeset a list of changes made in each version of the package code. It is customary to include an entry for the first version of the code, as shown in line 57 of Figure 9. The first argument is the version number in which the change was made; the second argument is the date the change was made; and, the third argument is a description of the change. If `\changes` is called from within the description of a macro or environment, the change is associated with that macro or environment. Otherwise, the change is categorized as "General".

The `\GetFileInfo` macro (line 59) reads the given file and parses its invocation of `\ProvidesFile` (lines 20 and 25 of Figure 6). `\GetFileInfo` makes the date part of `\ProvidesFile`'s argument available as `\filedate`, the version as `\fileversion`, and the package description as `\fileinfo`. The documentation can then use those macros when referring to the package.

```
57 % \changes{v1.0}{2008/02/18}{Initial version}
58 %
59 % \GetFileInfo{mypackage.dtx}
60 %
61 % \DoNotIndex{\newcommand,\newenvironment}
```

Figure 9: Miscellaneous initialization commands

One of Doc's most useful features is the automatic production of a code index. Every control sequence defined or used by the package is automatically indexed. However, particularly common control sequences can be distracting and should be omitted from the index. The `\DoNotIndex` macro takes a comma-separated list of control sequences that should not be indexed. (`\DoNotIndex` can be — and usually is — invoked repeatedly, with one line's worth of control sequences at a time.) Typically, TeX primitives such as `\if` / `\else` / `\fi`, `\begingroup` / `\endgroup`, and `\def` / `\edef` / `\gdef` / `\xdef` appear as arguments to `\DoNotIndex`, as do common macros from the LaTeX kernel such as `\newcommand` / `\renewcommand` and `\newcounter` / `\newsavebox` / `\newlength`. However, a package that redefines `\newcounter`, for example, probably *would* want to index that control sequence. Producing a good index takes a lot of judgment; think carefully about what someone reading the code might be interested in locating.

```
62 % \title{The \textsf{mypackage} package\thanks{This document
63 %   corresponds to \textsf{mypackage}~\fileversion, dated \filedate.}}
64 % \author{Your Name Here \\ \texttt{you@yournamehere.org}}
65 %
66 % \maketitle
67 %
68 % \section{Introduction}
69 %
70                                   ⋮
71 %
72 % \section{Usage}
73 %
74                                   ⋮
75 %
76 % \DescribeMacro{\myMacro}
77 % This macro does nothing.\index{doing nothing|usage}  It is merely an example.  If this were a
78 % real macro, you would put a paragraph here describing what the macro is supposed to do, what
79 % its mandatory and optional arguments are, and so forth.
80 %
81 % \DescribeEnv{myEnv}
82 % This environment does nothing.  It is merely an example.  If this were a real environment, you
83 % would put a paragraph here describing what the environment is supposed to do, what its
84 % mandatory and optional arguments are, and so forth.
```

Figure 10: Prose description of the package

`\myMacro`	This macro does nothing. It is merely an example. If this were a real macro, you would put a paragraph here describing what the macro is supposed to do, what its mandatory and optional arguments are, and so forth.
`myEnv`	This environment does nothing. It is merely an example. If this were a real environment, you would put a paragraph here describing what the environment is supposed to do, what its mandatory and optional arguments are, and so forth.

Figure 11: Typeset output of `\DescribeMacro` and `\DescribeEnv`

3.5 User documentation

Package documentation usually begins with a few sections of documentation for the user of the package, as shown in Figure 10. The `\title` specification in lines 62 and 63 is fairly typical in that it sets the package name with `\textsf` and uses `\thanks` to include a footnote with the package's version number and release date. `\date` is often omitted from the title block to distinguish the date the document was printed (`\today`) from the date the package was last modified (`\filedate`).

There is no `\begin{document}` in Figure 10 because the `\begin{document}` already appeared in the `.dtx` driver code (Figure 7); the code in Figure 10 is included through the driver code's invocation of `\DocInput`.

It is common to begin the package documentation with an introductory section that describes what the package does and a usage section that explains how to use the package. The Doc package provides two macros that help give a uniform look to usage sections in package documentation: `\DescribeMacro` and `\DescribeEnv`. Figure 11 displays how Doc typesets lines 76–84 of Figure 10. Notice that the macro or environment name is placed in the margin, where it is easy for a reader to find. Furthermore, the macro/environment name is automatically indexed, with the corresponding page number appearing in the so-called `usage` style (normally italics) in the index. Line 77 of Figure 10 shows how to index arbitrary text in the same style, using `\index{`⟨*term*⟩`|usage}`.

3.6 Package source code

The documented package source code follows the user documentation. Because the average user is not interested in the package's implementation, Doc enables a user to avoid including the package's source code when building the documentation by inserting a call

```
85  % \StopEventually{}
86  %
87  % \section{Implementation}
88                               ⋮
89  % \begin{macro}{\myMacro}
90  % The |\myMacro| macro takes a person’s name
91  % and returns the string ‘‘Hello,
92  % \meta{name}’’.
93  %    \begin{macrocode}
94  \newcommand{\myMacro}[1]{%
95    Hello, #1\relax
96  }
97  %    \end{macrocode}
98  % \end{macro}
99                               ⋮
100 % \Finale
```

Figure 12: Sample Implementation section

to `\OnlyDescription` into the `.dtx` driver code (between the `\documentclass` and `\begin{document}` lines in Figure 7).

Figure 12 shows how to document the package’s source code. The entire code should be bracketed between a call to `\StopEventually` (line 85) and a call to `\Finale` (line 100). `\StopEventually` takes an argument, which is the text for all of the sections that follow the package source code, for example the list of references or the package’s copyright and license information. Because the text appears as an argument to a command, certain LaTeX constructs such as `\verb` cannot be used within `\StopEventually`. Unfortunately, ordinary document sections cannot simply be placed after the call to `\Finale` because `\OnlyDescription` would still discard them.

It is good practice to use the standard LaTeX sectioning commands within the implementation section to organize the code and clarify its structure; for example, `\subsection{Initialization macros}`, `\subsection{Helper macros}`, `\subsection{User-callable macros and environments}`, ... One of the beauties of literate programming is that any LaTeX code can be used to document a package: tables, figures, mathematics — whatever is appropriate for explaining how the package works.

Lines 89–98 of Figure 12 give a sample macro definition. A macro definition starts with `\begin{macro}` and the macro name and ends with `\end{macro}`. The Doc package puts the macro name in the margin and includes an index entry with the source-code line number set in the `main` style (normally underlined).

Following the `\begin{macro}` comes the description of what the macro does. The sample description in Figure 12 uses two convenient features of the Doc package. First, “`|`” toggles verbatim mode, which is convenient for macro documentation that would otherwise be cluttered with `\verb` invocations. (This shortcut is in fact provided by the shortvrb package, which is included by Doc.) One caveat is that “`|`” cannot then be used in a `tabular` (or other) environment without first disabling its verbatim properties using `\DeleteShortVerb` and reenabling them afterwards with `\MakeShortVerb`. See the Doc documentation [4] for more information. The second useful Doc feature that appears in Figure 12 is `\meta`, which typesets its argument in italics and within angle brackets, as in “⟨*name*⟩”. This is useful for typesetting metasyntactic variables such as ⟨*number*⟩ or ⟨*length*⟩.

The macro source code appears, uncommented, within a `macrocode` environment. Because of some behind-the-scenes trickery in how `macrocode` is handled, there must be *exactly* four spaces between the “`%`” and the `\begin{macrocode}` (as shown in line 93) and between the “`%`” and the `\end{macrocode}` (as shown in line 97). When the documentation is typeset, the lines between `\begin{macrocode}` and `\end{macrocode}` are automatically numbered, and all control sequences encountered are automatically indexed in an unadorned style.

While Figure 12 shows only how to define a macro, environments are defined analogously, using `\begin{environment}` / `\end{environment}` instead of `\begin{macro}` / `\end{macro}` but still using `\begin{macrocode}` / `\end{macrocode}` to delineate blocks of LaTeX code. Definitions of things other than macros and environments — lengths, counters, boxes, etc. — should be placed within a `macro` environment.

The sample macro definition given in Figure 12 is typical of short, simple macros. Longer, more complex macros may benefit from additional commentary within the macro body. In addition, it is common in LaTeX for macros to define other macros. A `.dtx` file can handle both of these cases: Figure 13 shows how. It may be easier to follow Figure 13 by comparing it to the typeset output, shown in Figure 14. Notice how the `\begin{macro}{\othermacro}` is nested within the `\begin{macro}{\complexdef}`. Packages that include a number of short, related definitions (e.g., a set of `\newlength` calls) commonly specify a sequence of `\begin{macro}` calls followed by a description of all the definitions as a whole (e.g., “These lengths represent the jabberwock’s width, height, and depth”), followed by a single `macrocode` environment that includes all of the related declarations back-to-back.

```
% \begin{macro}{\complexdef}
% This is a more sophisticated use of the |macro| and |macrocode| environments than was used in
% Figure 12.  Notice the nested |macro| environments and the repeated |macrocode| environments.
% \changes{v1.1}{2008/02/18}{Changed ‘‘Goodbye’’ to ‘‘Hello’’}
%    \begin{macrocode}
\DeclareRobustCommand{\complexdef}[1]{%
  Hello, #1.
%    \end{macrocode}
% You can insert comments anywhere.  Just call |\end{macrocode}|, enter your text, and start a
% new |\begin{macrocode}|.
%    \begin{macrocode}
  How do you like my macro?%
%    \end{macrocode}
% \begin{macro}{\othermacro}
% Here we have the |\othermacro| macro defined within the |\complexdef| macro.  |macro|
% environments are allowed to nest.
%    \begin{macrocode}
  \gdef\othermacro{#1}%
}
%    \end{macrocode}
% \end{macro}
% \end{macro}
```

Figure 13: A more complex macro definition

`\complexdef` This is a more sophisticated use of the `macro` and `macrocode` environments than was used in Figure 12. Notice the nested `macro` environments and the repeated `macrocode` environments.

```
\DeclareRobustCommand{\complexdef[1]{%
  Hello, #1.
```

You can insert comments anywhere. Just call `\end{macrocode}`, enter your text, and start a new `\begin{macrocode}`.

```
  How do you like my macro?%
```

`\othermacro` Here we have the `\othermacro` macro defined within the `\complexdef` macro. `macro` environments are allowed to nest.

```
  \gdef\othermacro{#1}%
}
```

Figure 14: Typeset version of Figure 13

3.7 The change history and index sections

The `\changes` call in Figure 13 is not typeset in place but rather schedules a line to be added to the document's Change History section. Because Figure 13's `\changes` call appears within a `macro` environment it is assumed to apply to the surrounding macro instead of to the document as a whole. Figure 15 illustrates how the Change History section may appear in the typeset documentation. If `\changes` appears outside of a `macro` or `environment` environment, the corresponding line in the Change History section lists "General" in place of a macro/environment name.

Running the `.dtx` file through `latex` produces a corresponding `.idx` file if `\CodelineIndex` appears in the driver code and a corresponding `.glo` file if `\RecordChanges` appears in the driver code. The `makeindex` program [2] can be used as shown in

Change History

v1.1
  `\complexdef`: Changed "Goodbye" to "Hello" 1

Figure 15: Sample Change History section

```
makeindex -s gind.ist -o ⟨package⟩.ind \
  ⟨package⟩.idx
makeindex -s gglo.ist -o ⟨package⟩.gls \
  ⟨package⟩.glo
```

Figure 16: Commands for producing an index and a change history

Figure 16 to convert the `.idx` file to a typeset index (`.ind`) and the `.glo` file to a typeset change history (`.gls`).

3.8 Additional notes about comments

Program comments should not be written between `\begin{macrocode}` and `\end{macrocode}` because everything within a `macrocode` environment is typeset as code, not as formatted text. (Figure 13 shows the proper way to include inline code comments.) However, it is possible to write comments that are not typeset at all (e.g., for documenting a macro definition that is part of the user documentation, not of the package itself), In fact, all combinations of "visible in the user documentation" and "visible in the `.sty` file" are possible. Table 1 summarizes the techniques for achieving each of these combinations.

Table 1: Comment visibility

Appears in docs	Appears in `.sty`	Mechanism
N	N	`% ^^A` ⟨*comment*⟩

N	Y	`% \iffalse`
		`%%` ⟨*comment*⟩
		`% \fi`

Y	N	`%` ⟨*comment*⟩
Y	Y	`%%` ⟨*comment*⟩

4 Concluding remarks

The advantage of using `.ins` and `.dtx` files is that they encapsulate not only the LaTeX-readable package code but also a human-readable description of the code. Unlike typical, text-only program comments, documentation produced from `.ins` and `.dtx` files can take advantage of all of LaTeX's typesetting power — sectioning, cross-references, figures, tables, mathematics, etc. — coupled with automatic indexing of all macro and environment definitions and uses and automatically pretty-printed code listings. Because of their ability to facilitate the production of immensely readable package documentation, `.ins` and `.dtx` files are the most popular way to distribute LaTeX packages and represent a technique that all LaTeX package writers should strongly consider using for their own packages.

References

[1] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984. Available from `http://www.literateprogramming.com/knuthweb.pdf`.

[2] Leslie Lamport. *MakeIndex: An Index Processor for LaTeX*, February 17, 1987. Available from `http://www.ctan.org/get/indexing/makeindex/doc/makeindex.pdf`.

[3] Frank Mittelbach. The `doc`—option. *TUGboat*, 10(2):245–273, July 1989. Available from `http://www.tug.org/TUGboat/Articles/tb10-2/tb24mitt-doc.pdf`.

[4] Frank Mittelbach. The `doc` and `shortvrb` packages. Distributed as part of LaTeX 2_ε, February 9, 2004. Document source is available from `http://www.ctan.org/get/macros/latex/base/doc.dtx`.

[5] Frank Mittelbach, Denys Duchier, Johannes Braams, Marcin Woliński, and Mark Wooding. The DocStrip program. Distributed as part of LaTeX 2_ε, July 29, 2005. Available from `http://www.ctan.org/get/macros/latex/base/docstrip.dtx`.

[6] The LaTeX3 Project. LaTeX 2_ε for class and package writers. Distributed with LaTeX 2_ε as `clsguide.dvi`, February 15, 2006. Also available from `http://www.ctan.org/get/macros/latex/doc/clsguide.pdf`.

◇ Scott Pakin
4975 S. Sol
Los Alamos, NM 87544-3794
USA
`scott+tb (at) pakin dot org`
`http://www.pakin.org/~scott`