The `Document` Package

A Simplified Approach to Literate Programming

Michael L. Hall

May 26, 2004

Abstract

The `Document` package is used to process documentation that is imbedded in the comments of a source code file. Any source code language may be used, if the proper values for comment characters have been set. Any formatting language may be used as well. Currently, options for Fortran, M4, C++ (or C), LaTeX, Java, Prolog and shell scripts (perl, sh, csh, awk, Tcl) as the source code language have been implemented.

User's Note: the `-docstyle` keyword which enabled compatibility with the doc routine has been deselected. `Document` no longer includes this capability.

Laziness: The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it. Hence, the first great virtue of a programmer. Also, hence this book [program -MLH].

Impatience: The anger you feel when the computer is being lazy. This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least that pretend to. Hence, the second great virtue of a programmer.

Hubris: Excessive pride, the sort of thing that Zeus zaps you for. Also the quality that makes you write (and maintain) programs that other people won't want to say bad things about. Hence, the third great virtue of a programmer.

From the glossary of *Programming perl*, by Larry Wall and Randall L. Schwartz.

Table of Contents

1 Author, Date and Version Information

Filename:	Document
Author:	Michael L. Hall Los Alamos National Laboratory P.O. Box 1663, MS-D409 Los Alamos, NM 87545 Email: `hall@lanl.gov`
CVS Info:	`Id: Document,v 12.9 2004/02/06 17:41:42 hall Exp`
Contributions:	While the entire code (including any bugs) is solely the responsibility of the author, the following people were involved in discussions that proved very helpful: Mark Gray, Chuck Henkel and John Turner.

2 Quick Reference

Document Keywords for -generic Option

Keyword	Meaning
Begin_Doc [filename]	Start Documentation Mode and output to filename if specified.
End_Doc	Stop Documentation Mode.
Begin_Verbatim	Start Verbatim Mode (only during Documentation Mode) to leave comment characters as they appear in the input.
End_Verbatim	Stop Verbatim Mode.
Begin_Self_Documentation	Start Self Documentation Mode, which lists the commands necessary to produce the documentation for this file. This may or may not appear within the Documentation Mode.
End_Self_Documentation	Stop Self Documentation Mode.
Begin_Self_Test	Start Self Test Mode, a synonym for the Self Documentation Mode.
End_Self_Test	Stop Self Test Mode, a synonym for Self Documentation Mode.

Document keywords must appear on a comment line by themselves to take effect. Lines containing Document keywords are never printed.

Common syntax for including source code (with LaTeX and Fortran):

```
c Begin_Doc
c \begin{verbatim}
c Begin_Verbatim
c
c Source Code to be quoted verbatim.
c
  do i = 1, 20
    i = i + 1
  end do
c End_Verbatim
c \end{verbatim}
c End_Doc
```

The perl source code for the Document package contains examples of most of the things that can be done with Document, LaTeX and LaTeX2HTML, such as:

- Self-Documentation scripts
- Multiple output files and Document options
- Included code
- Indexing
- Items included in only LaTeX or HTML
- Hyper references
- HTML lists and Exterior HTML links
- Verbatim environments from hell

---

3 Usage

3.1 Generating Documentation for Document

To generate the documentation for this routine, you can use Document's self-documentation option to run an internally-defined script:

```
% Document -self Document
```

Or, to generate the documentation manually, run Document on itself: [1]

```
% Document Document
% cd Document_doc
% latex Document
% makeindex Document
% latex Document
% latex Document
```

To view the dvi file, continue with:

```
% xdvi Document_doc/Document &
```

If PostScript output is desired, then continue with:

```
% dvips Document
% lpr Document.ps
```

If HTML documentation is desired, then continue with:

```
% cp Document.dvi ..
% latex2html -nomath -html_version 4.0,math -no_subdir -toc_stars Document
% mv ../Document.dvi .
% cp ../Document .
% cp ../hallstone.jpg .
% cp -r /usr/share/latex2html/icons .
% rm Document.tar.gz *.aux *.idx *.ilg *.ind *.log *.toc
% cd ..
% gtar cvzhf Document.tar.gz Document_doc
% mv Document.tar.gz Document_doc
```

and load the URL "file://localhost/<current_directory>/Document_doc/index.html" into your favorite web browser.

[1]To make the documentation look exactly like the official version, you will need the official .latex2html-init file.

3.2 Software Requirements

To run `Document`, you must have `perl` installed. You may also need a compiler if you want to process the source code, and a formatter if you want to process the `Document` output (documentation) file.

To generate the documentation for `Document` in PostScript, you will need LaTeX 2_ε, `makeindex` and `dvips`. To generate the documentation in HTML you will need LaTeX 2_ε, `makeindex`, gnu tar, and LaTeX`2HTML`, which requires `ghostscript`, `netpbm`, and `perl`, version 5.

The HTML version will look better if LaTeX`2HTML` v.98.2beta8, (or later) is used.

3.3 Running the Script

Once the source code with the documentation imbedded in the comments has been produced, it can be processed by `Document` in any of the following ways:

```
% Document [options and files] > outfile
% cat file1 [file2, etc.] | Document [options] > outfile
```

The options and files may be in any order on the command line. The files are processed in the command line order. The available options are:

Source Language and Formatting Language Options

Option	Language	Meaning
`-c++`	Source Code	Use comment characters and other commands suitable for processing a C++ or C source code file.
`-generic`	Formatting	Use verbatim definitions and other commands suitable for processing a file with generically formatted comments. This is the default (and recommended) setting. It is required if comment characters are to be stripped off of verbatim environments.
`-fortran`	Source Code	Use comment characters and other commands suitable for processing a Fortran source code file.
`-java`	Source Code	Use comment characters and other commands suitable for processing a Java source code file.
`-latex`	Source Code	Use comment characters and other commands suitable for processing a LaTeX *source code* file.
`-m4`	Source Code	Use comment characters and other commands suitable for processing an m4 source code file.
`-prolog`	Source Code	Use comment characters and other commands suitable for processing a Prolog source code file.
`-shell`	Source Code	Use comment characters and other commands suitable for processing a shell script (csh, sh, awk, perl).

General Options

Option	Meaning
`-blanks=number`	The number of blanks to match and remove after comment keywords. The default is one.
`-dir=dirname`	Output any named files to the specified directory.
`-headeronly`	Ignore any `Document` keywords in the file and output the file header only, verbatim, to standard out. The file header consists of all lines that are comment lines at the beginning of a file – the first non-commented line marks the end. If there are files named .doc_header and .doc_footer located in the current directory, they will be prepended and appended to the output. See the Unmodified Files section (section 3.5) for more information.
`-self`	Use the self-documentation mode which extracts a shell script from the file and runs it. Note that a source code language must still be specified (or defaulted) so that the proper comment characters are used.
`-selftest`	Use the self-test mode which extracts a shell script from the file and runs it. This is a synonym for the `-self` flag.
`-silent`	Do not output anything to STDERR. This includes informational output. Also, do not query the user for verification before executing scripts.
`-verbatim`	Ignore any `Document` keywords in the file and output the entire file, verbatim, to standard out. If there are files named .doc_header and .doc_footer located in the current directory, they will be prepended and appended to the output. See the Unmodified Files section (section 3.5) for more information.

The `Document` package can be used with any formatting language, and can be easily extended to other source code languages.

If an option is specified for the source code language on the command line, then it applies to all of the files being processed. If no source code language option is given on the command line, then the first twenty lines of each file are checked for a line containing:

```
Document options: [options]
```

which will specify options for that file only. If the source code language is still undefined, then the directory that contains the input file is searched for a file named .doc_options, which contains a single line with only the desired options. If the source code language is still undefined, the suffix of the file type will be checked for matches with defined suffix types for each source code language:

Source Code Language	File Suffixes
C++	.C, .cc, .c, .H, .hh, .Cpp, .cpp, .hxx, .cxx
Fortran	.F, .f, .h, .F90, .FCM, .inc, .fm4
Java	.java
LaTeX	.tex
M4	.m4, .gm4
Prolog	.ari, .pro, .nl
Shell	.awk, .pl, .perl, .sed, .tcl, no suffix

If the source code language is still undefined, then it will default to `-fortran`.

3.4 File Format

The idea behind the `Document` package is that all documentation keywords are placed in the comments of the source code document being processed. The source code can then be processed by the compiler or interpreter without additional changes. The `Document` keywords instruct the `Document` package how to extract documentation from the source code file without affecting the operation of the source code file itself.

`Document` keywords are different for each command line option that is specified. In this description, the `Document` keywords for the default options (`-generic` and `-fortran`) will be used, with the name of the perl list variable for the specific keyword in parantheses. For the values of the keywords for other options, consult the Coding Details section (see section 5).

To start the documentation process, enter the `Begin_Doc (start_documentation)` keyword, on a line by itself, in the comments of the file:

```
c This line of text will not be in the documentation file.
c Begin_Doc
c This line of text will be entered into the documentation file.
```

To terminate the documentation process, enter the `End_Doc (end_documentation)` keyword, on a line by itself, in the comments of the file:

```
c This line of text will be entered into the documentation file.
c End_Doc
c This line of text will not be in the documentation file.
```

The format given above will send the documentation output to standard output. An optional filename may be appended to the `Begin_Doc` keyword to redirect output. This filename will be valid only for the current documentation block. If the same filename is specified more than once, the first specification will open the file and subsequent specifications will append to the file. This feature can be used to change the order of input for the documentation, as is shown in this example:

```
c Begin_Doc main.tex
c % Note that the order of files a.tex and b.tex has been switched.
c \input{b}
c \input{a}
c End_Doc
c
c Begin_Doc a.tex
c This line is in file a.tex.
c End_Doc
c
c Begin_Doc b.tex
c This line is in file b.tex.
c End_Doc
c
c Begin_Doc a.tex
c This line is appended to file a.tex.
c End_Doc
```

The default documentation mode removes the comment characters from the documentation before printing. For Fortran, the comment character keywords (`comment_characters`) are either a "c" or whitespace and an exclamation point ("!") at the beginning of the line. In-line comments with coding followed by an exclamation point and a comment are not modified. For example, in the default documentation mode these lines:

```
c Begin_Doc
c
c Comment.
c
      ! Bang-style comment.
       do i = 1, 10
         a(i) = i**2  ! In-line comment
       end do
c End_Doc
```

would appear like this in the output:

```

 Comment.

 Bang-style comment.
       do i = 1, 10
         a(i) = i**2  ! In-line comment
       end do
```

To stop the removal of comment character keywords from the output (i.e. to leave lines unmodified), surround the text with the `Begin_Verbatim (start_verbatim)` keyword, on a line by itself, and the `End_Verbatim (end_verbatim)` keyword, on a line by itself, in the comments of the file:

```
c Begin_Doc
c These lines will have
c their comment characters removed
c
c \begin{verbatim}
c Begin_Verbatim
c
c But these lines won't.
c
c End_Verbatim
c \end{verbatim}
c End_Doc
```

which will appear as follows in the output:

```
 These lines will have
 their comment characters removed

 \begin{verbatim}
c
```

```
c But these lines won't.
c
 \end{verbatim}
```

Notice the use of the LaTeX verbatim environment on the outside of the `Document` verbatim keywords. This is a common construct for situations where quoting code and preserving the comments is desired.

Two last `Document` keywords are `always_print` and `never_print`. If any line is found with either of these keywords (even if there are other things on the line), the specified action will be taken. The `never_print` keyword takes precedence, and is primarily used for the self-documentation option (see the Self-Documentation section, section 3.6). For `always_print`, comments will be stripped off unless currently inside a verbatim environment. `always_print` is not used with either the `-generic` or `-fortran` options.

3.5 Running `Document` on Unmodified Files

To reap the full benefits of using the `Document` package, one must make modifications to the source code files. However, there are situations where it is desireable to have some form of rudimentary documentation for a package that for some reason has unmodified source. For instance, it could be source that is not maintained by the person doing the documentation, or it could be the first attempt at using `Document` before making specialized modifications to the source.

`Document` provides two options that are useful in this situation, `-verbatim` and `-headeronly`. The `-verbatim` option outputs the entire contents of the file with no changes to standard out. The `-headeronly` option outputs only the header of the file, which is defined as every line until a non-comment line is reached. Additionally, both options will prepend and append the files .doc_header and .doc_footer if they exist. The .doc_header and .doc_footer files can contain the following strings which will have the correct values substituted in:

String	Substitution
doc_filename	The complete filename, without the directory.
doc_filename_base	The filename without any suffix, without the directory.
doc_dirname	The immediate directory name (not the full directory name, but only the directory name one level up from the input file).

The header and footer files may be configured for use with any formatting language. A common choice for the header and footer files for use with LaTeX and LaTeX2HTML is:

.doc_header:

```
\subsection{doc_filename_base \label{doc_filename_base}}
\index{doc_dirname!doc_filename}
\begin{verbatim}
```

.doc_footer:

```
\end{verbatim}
```

The desired options for processing the files (for example, the `-verbatim` or the `-headeronly` option) may be specified in a .doc_options file that is placed in the same directory.

3.6 Self-Documentation

The commands used to generate the documentation for a file can be imbedded in the file in a similar manner to the documentation itself. To activate this feature, use the `-self` option. For example, the documentation for document can be generated by typing:

```
% Document -self -shell Document
```

The `-self` option parses the input file and removes comment characters according to the setting of the source code option. The `start_documentation` keyword is set to `Begin_Self_Document`, and the `end_documentation` keyword is set to `End_Self_Document`. The `-self` option then removes any initial "%" sign (often used to indicate shell input), and executes each line in turn.

Here is an example self-documentation entry, in a Fortran code segment:

```
c Begin_Self_Document
c   % Document -fortran routine.F > routine.tex
c   % latex routine
c End_Self_Document
c
c  Other code and comments.
c
c Begin_Self_Document
c   % makeindex routine
c   % latex routine
c   % latex routine
c   % dvi routine
c   % latex2html -dir routinehtml -nomath -html_version 4.0,math routine
c End_Self_Document
```

which would execute the following commands:

```
Document -fortran routine.F > routine.tex
latex routine
makeindex routine
latex routine
latex routine
dvi routine
latex2html -dir routinehtml -nomath -html_version 4.0,math routine
```

The `-self` option is equivalent to the `-selftest` option, and the beginning and ending keywords can be written as `Begin_Self_Test` and `End_Self_Test`. This usage underscores that the self-documentation (or self-test) feature is a very powerful feature with many uses – its primary function is simply to couple a script with some input data so that they can be maintained in sync. The function could be

- processing the documentation in a file according to a script contained in the same file;
- compiling, running, and examining the output of the source code in a file according to a script contained in the same file; or

- writing several input files, running an executable on them, and examining the output of the run, with all of the input files and the execution script contained in the same file.

4 Comparisons to Other Literate Programming Packages

4.1 Comparison with WEB

The `Document` package is much less complex than WEB and the various WEB-like and WEB-derivative packages. The entire source and documentation for `Document` is around 1000 lines, compared to 10,000 lines for the original WEB. Other differences between `Document` and the sea of WEB style programs are: [2]

Input File: Most WEB packages start from an input file that is not a valid source code file. A program called `weave` takes the input file and makes a formatting language file, while a program called `tangle` takes the input file and makes a source code file. The `Document` package takes an input file that is already a valid source code file. The formatting language commands are imbedded in the comments of the source code, so that no changes need to be made to the file for compilation.

Macros: Most WEB packages allow the user to define macros that stand for chunks of code or documentation, and that can be used to move code around and alter the flow of the program. The `Document` package does not provide this feature, and therefore the source code must be in the order in which it is to be compiled. There is, however, a mechanism to print out multiple documentation files. When combined with a formatting language that provides for the inclusion of files, this allows the user to alter the structure of the documentation to his own needs.

Languages: Many, but not all, of the WEB packages are language-specific (both formatting and source code). For example, the original WEB package only worked with Pascal and TEX. `Document` is only slightly language-specific in that it needs to know something about the syntax (mainly the comment characters) of the source code language. It can be easily extended to languages that are not already incorporated.

Indexing: Many WEB packages provide automatic indexing of things like variables and routines. `Document` does not do indexing automatically, but it can be done if the formatting language supports it, as is possible with LATEX and LATEX2HTML.

For more information on Literate Programming, a good starting point is the Literate Programming FAQ located at `ftp://rtfm.mit.edu/pub/usenet/news.answers/literate-programming-faq`.

4.2 Comparison with `doc`

In the past, the `Document` package was able to process source code formatted for the `doc` routine (written by Mark Gray), but this capability has been de-selected. This section remains for historical reasons, but will be deleted eventually.

Even when operating in the "docstyle" option, there might have been slight differences between the output of the two routines:

Keywords: The `Document` package requires that keywords be on a line by themselves (after removing comments from the line), except for the `always_print` keywords. The `doc` routine does not have this requirement.

[2] As a caveat I will first state that I have never used WEB or any of the WEB-like or WEB-derivative packages. I have read the FAQs and documentation, and looked at sample code to the extent that I have a reasonably good idea of their capabilities.

In addition to these differences, there are some other differences between `Document` and `doc`:

Self-Documentation: `Document` allows the commands necessary to process the documentation to be imbedded in the file as well as the documentation itself. This feature can also be used to include commands which will self-test or run the source code.

Multiple Output Files: `Document` allows output to multiple files from a single source file. This allows the user to re-arrange the structure of the documentation to suit his purposes.

Language Options: `Document` has options which allow the user to specify the source code language.

Extensibility: `Document` can easily be extended to include new source code languages. The `doc` routine is somewhat restricted to a single formatting language, whereas `Document` is independent of formatting language.

Philosophy: `Document` makes a distinction between the keywords (i.e. the language of the `Document` package), the source code language, and the formatting language. The `doc` routine does not make this distinction, and uses elements of the source code language and the formatting language as keywords.

5 Coding Details

```
# Settings.

$, = '';                     # Set output field separator.
$\ = "\n";                   # Set output record separator.
$true = "1";                 # Name for true variable.

# Save STDOUT for later use.

open(SAVESTDOUT,">&STDOUT");

# Process command line options.

&set_options(@ARGV);
($source) && ($default_source = $source);
($format) && ($default_format = $format);
($file_verbatim) && ($default_file_verbatim = $file_verbatim);
($header_only) && ($default_header_only = $header_only);

# Subroutine to process options.

sub set_options {
    foreach (@_) {
        /^(-generic)$/ && ($format = $_);
        /^(-c\\+\\+|-fortran|-java|-latex|-m4|-prolog|-shell)$/ && ($source = $_);
        /^-self$/ && ($self_doc = $true);
        /^-selftest$/ && ($self_doc = $true) && ($self_test = $true);
        /^-silent$/ && ($silent = $true);
        /^-dir=(\S*)$/ && ($directory = $1);
        /^-blanks=(\S*)$/ && ($number_of_blanks = $1);
        /^-verbatim$/ && ($file_verbatim = $true);
        /^-headeronly$/ && ($header_only = $true);
    }
    !$silent && !$self_doc && ($verbose = $true);
}

# Sort argument list to put flags in front, then remove them.

@ARGV = sort @ARGV;
while ($ARGV[0] =~ /^-/) {shift;}

# Loop over input files.

foreach $infile (@ARGV) {
    open(INFILE, "$infile") || die "Can't open $infile: $!";

    # Set default languages and modes from command line options.

    $format = 0;
    $source = 0;
    $file_verbatim = 0;
```

```
    $header_only = 0;
    ($default_format) && ($format = $default_format);
    ($default_source) && ($source = $default_source);
    ($default_file_verbatim) && ($file_verbatim = $default_file_verbatim);
    ($default_header_only) && ($header_only = $default_header_only);

    # Read first lines of file for options if necessary.

    if (!$source) {
$i = 0;
        while ($i < 20) {
            $i++;
            $_ = <INFILE>;
            if (/[dD]ocument\s*[oO]ptions:\s*(.*)$/) {
                &set_options(split(/ /,$1));
            }
        }
        seek(INFILE,0,0);   # Rewind the file.
    }

    # Look for .doc_options file for options if necessary.

    ($doc_options = $ENV{"PWD"}."/".$infile) =~ s|/[^/]*$||;
    $doc_options .= "/.doc_options";
    if (!$source) {
        if (-r $doc_options) {
            open(OPTIONS, "$doc_options") || die "Can't open $doc_options: $!";
            $_ = <OPTIONS>;
            &set_options(split);
            close(OPTIONS);
        }
    }

    # Set source code language by file suffix if necessary.

    if (!$source) {
        $_ = $infile;
        if (/\.([^\.]*)$/) {
            $_ = $1;
        }
        else {
            $_ = "";
        }
        /^(C|cc|c|H|hh|Cpp|cpp|hxx|cxx)$/ && ($source = "-c++");
        /^(F|f|h|F90|FCM|inc|fm4)$/ && ($source = "-fortran");
        /^(java)$/ && ($source = "-java");
        /^(tex)$/ && ($source = "-latex");
        /^(m4|gm4)$/ && ($source = "-m4");
        /^(ari|pro|nl)$/ && ($source = "-prolog");
        /^(awk|pl|perl|sed|sh|tcl)$/ && ($source = "-shell");
        # Uncomment next line to make .pl files default to Prolog.
        #/^(pl)$/ && ($source = "-prolog");
        ($_ eq "") && ($source = "-shell");
    }
```

```
# Set last resort default languages if necessary.

(!$source) && ($source = "-fortran");
(!$format) && ($format = "-generic");
```

```
# Lists of reserved words related to the formatter.
#
#   -generic  : use generic commands (default).

if ($format eq '-generic') {
    @start_verbatim = ('Begin_Verbatim');
    @end_verbatim = ('End_Verbatim');
}
```

```
# Lists of reserved words related to the source code language.
#
#   -c++      : use C++ (or C) commands.
#   -fortran  : use Fortran commands (default).
#   -java     : use Java commands (same as C++).
#   -latex    : use LaTeX commands (as a *source* code language).
#   -m4       : use m4 commands.
#   -prolog   : use Prolog commands.
#   -shell    : use shell (csh, sh, perl, awk, tcl) commands.

if ($source eq '-c++') {
    @start_documentation = ('Begin_Doc');
    @end_documentation = ('End_Doc');
    @always_print = (@always_print);
    @comment_characters = ('^\s*//','^\s*/\*','^#');
    @comment_endings = ('\*/');
}
elsif ($source eq '-fortran') {
    @start_documentation = ('Begin_Doc');
    @end_documentation = ('End_Doc');
    @always_print = (@always_print);
    @comment_characters = ('^[Cc]','^\\s*!');
}
elsif ($source eq '-java') {
    @start_documentation = ('Begin_Doc');
    @end_documentation = ('End_Doc');
    @always_print = (@always_print);
    @comment_characters = ('^\s*//','^\s*/\*\*','^\s*/\*');
    @comment_endings = ('\*/');
}
elsif ($source eq '-latex') {
    @start_documentation = ('Begin_Doc');
```

```
    @end_documentation = ('End_Doc');
    @always_print = (@always_print);
    @comment_characters = ('^\s*%');
}
elsif ($source eq '-m4') {
    @start_documentation = ('Begin_Doc');
    @end_documentation = ('End_Doc');
    @always_print = (@always_print);
    @comment_characters = ('^\s*dnl');
}
elsif ($source eq '-prolog') {
    @start_documentation = ('Begin_Doc');
    @end_documentation = ('End_Doc');
    @always_print = (@always_print);
    @comment_characters = ('^\s*%','^\s*\/\*');
    @comment_endings = ('\*/');
}
elsif ($source eq '-shell') {
    @start_documentation = ('Begin_Doc');
    @end_documentation = ('End_Doc');
    @always_print = (@always_print);
    @comment_characters = ('^\s*#');
}




# Self-Documentation (or Self-Test) Mode.

if ($self_doc) {
    @always_print = ();
}
@never_print = (@never_print,'Begin_Self\_Document','End_Self\_Document');
@never_print = (@never_print,'Begin_Self\_Test','End_Self\_Test');




# Verbatim and Header-Only Modes.

$doc_header = ".doc_header";
$doc_footer = ".doc_footer";
if ($file_verbatim || $header_only) {
    $include = $true;
    $verbatim = $true;
    ($infile_dir = $ENV{"PWD"}."/".$infile) =~ s|/[^/]*$||;
    $infile_dir =~ s|.*/||;
    ($infile_name = $infile) =~ s|^.*/||;
    ($infile_base = $infile_name) =~ s/\.[^\.]*$//;
    if (-r $doc_header) {
        open(HEADER, "$doc_header") || die "Can't open $doc_header: $!";
        while (<HEADER>) {
            chop;
```

```
            s/doc_filename_base/$infile_base/;
            s/doc_filename/$infile_name/;
            s/doc_dirname/$infile_dir/;
            print;
        }
        close(HEADER);
    }
}
else {
    $include = 0;
    $verbatim = 0;
}

# Set regexp for matching blanks after the comment keywords.
# Default number of blanks to match is one.

($number_of_blanks) || ($number_of_blanks = 1);
$blanks = '('.(' 'x$number_of_blanks).')?';

# Make regular expressions from the lists of keywords.

$comment_characters  = join("$blanks|",@comment_characters).$blanks;
if (@comment_endings) {
    $endl = '\s*('.join('|',@comment_endings).')*\s*$';
}
else {
    $endl = '\s*$';
}
$beginl = '^('.join('|',@comment_characters).')*'.$blanks.'\s*(';
$fname               = ')\s*([\S]*)'.$endl;
$start_documentation = $beginl.join('|',@start_documentation).$fname;
$end_documentation   = $beginl.join('|',@end_documentation).')'.$endl;
$start_verbatim      = $beginl.join('|',@start_verbatim)   .')'.$endl;
$end_verbatim        = $beginl.join('|',@end_verbatim)     .')'.$endl;
$always_print        = join('|',@always_print);
$never_print         = join('|',@never_print);

# Loop over lines from each input file.

while (<INFILE>) {
    chop;                 # Strip record separator.

    # Turn off documentation mode if one of the end_documentation
    # keywords is found. Only check if outside of verbatim scope.

    if (!$verbatim) {
        /$end_documentation/ && ($include = 0);
    }

    # Turn off verbatim mode if one of the end_verbatim
    # keywords is found (unless -verbatim or -headeronly
    # has been specified as a flag). Set the suppress
    # variable to suppress printing of the keyword line.
```

```
if (/$end_verbatim/ && !$file_verbatim && !$header_only) {
    $verbatim = 0;
    $suppress = $true;
}

# Turn off include mode at the end of the header for
# -headeronly option.

!/$comment_characters/ && ($header_only) && ($include = 0);

# Optional massaging of documentation.

if (!$verbatim) {

    # Remove comment characters.

    s/$comment_characters//;
}

# Turn on verbatim mode if one of the start_verbatim
# keywords is found. Set the suppress variable to
# suppress printing of the keyword line.

if (/$start_verbatim/) {
    $verbatim = $true;
    $suppress = $true;
}

# Suppress printing if a never_print keyword is found.

/$never_print/ && ($suppress = $true);

# Optional print.

if ($include && !$suppress) {

    # Self-Documentation (or Self-Test) mode.

    if ($self_doc) {

        # Remove Self-Doc Comment Characters.

        s/^\s*%//;
        push (@self_doc_script,$_);
    }

    # Regular print.

    else {
        print;
    }
}
else {
    $suppress = 0;
```

```
            # Print lines that contain an always_print keyword.

            ($#always_print != -1) && /$always_print/ && (print);
        }

        # Turn on documentation mode if one of the start_documentation
        # keywords is found. Start with verbatim turned off.

        if (/$start_documentation/ && !$header_only) {
            $include = $true;
            $verbatim = 0;

            # Redirect output as directed.

            $file = $4;
            if ($directory) {
                $file = $directory.'/'.$file;
                (! -d $directory) && (system "mkdir $directory");
            }
            close(STDOUT);
            if ($file eq "") {
                ($verbose) && (print STDERR "Output to STDOUT.");
                open(STDOUT,">&SAVESTDOUT");
            }
            elsif ($opened{$file}) {
                ($verbose) && (print STDERR "Output appended to ", $file, ".");
                open(STDOUT,">>$file");
            }
            else {
                ($verbose) && (print STDERR "Output to ", $file, ".");
                $opened{$file} = 1;
                open(STDOUT,">$file");
            }
        }
    }

    # Verbatim and Header-Only Modes.

    if ($file_verbatim || $header_only) {
        if (-r $doc_footer) {
            open(FOOTER, "$doc_footer") || die "Can't open $doc_footer: $!";
            while (<FOOTER>) {
                chop;
                s/doc_filename_base/$infile_base/;
                s/doc_filename/$infile_name/;
                s/doc_dirname/$infile_dir/;
                print;
            }
            close(FOOTER);
        }
    }

}
```

```
# Info to user.

$, = ' ';     # Set output field separator.
if ($verbose && keys(%opened)) {
    (print STDERR "Document wrote these files:", keys(%opened));
}

# Run Self-Documentation (or Self-Test) Script.

if ($self_doc) {
    if ($silent) {
        $answer = 'y';
    }
    else {
        if ($self_test) {
            print STDERR "The self-test option will execute this script:";
        }
        else {
            print STDERR "The self-documentation option will execute this script:";
}
        foreach $statement (@self_doc_script) {
            print STDERR $statement;
        }
        $\ = " ";
        print STDERR "Okay to proceed?";
        $answer = <STDIN>;
        $answer = substr($answer,0,1);
    }
    if ($answer eq 'y' || $answer eq "Y") {
        open(SCRIPT, ">/tmp/$$");
        $\ = "\n";
        foreach $statement (@self_doc_script) {
            print SCRIPT $statement;
        }
        close(SCRIPT);
        open(STDOUT,">&SAVESTDOUT");
        system "chmod a+x /tmp/$$; /tmp/$$; rm /tmp/$$";
    }
}

# Close SAVESTDOUT to eliminate *incorrect* warning message:
#
#   Name "main::SAVESTDOUT" used only once: possible typo.
#
# It's incorrect because SAVESTDOUT is used multiple times,
# e.g. about 10 lines above this comment.

close(SAVESTDOUT);
```