LaTeX Dogwagger

[A different approach to documentation]

Version 4.0.5

J.M. van Schalkwyk*

July 21, 2020

Contents

*With some help! See the change log.

1 Introduction

In a single line, you can change for the better how you document your computer code. Here's that line:

```
% DogWagger version='4.0.5' fileTarget='MYFILE.pl'
```

Inserted at the start of a LaTeX file, this comment line allows you to combine your code and documentation *without* having to learn the use of new tools. This example generates Perl but, with a little help from Dogwagger, a single LaTeX file can spawn multiple different files in multiple languages — html, PHP, Javascript, C, C++, SQL, R, Erlang, and so forth.

What's the catch?

The only catch is that you must be familiar with LaTeX, or (for those who are allergic to natural rubber), you must have tried the almost-human-friendly alternative, LyX.[1] If you're terrified by the idea of running a program from the command line, then Dogwagger may also not be for you.

Dogwagger is freely available under the GNU Public Licence (See Appendix A), and will work with a simple, free text editor (like Notepad, vim, Emacs) or specific LaTeX-friendly editors like LyX and Kile.

How does it work?

You simply embed all of your code within the LaTeX document as *verbatim* statements, and submit the .tex file to DogWagger, which pulls out the code and turns it into a program. The living proof of this is the document you're reading, which not only describes but *contains* the entire source code of Dogwagger!

Why Dogwagger?

It's simple. Other documentation managers have problems:

- Complex dependencies;
- A steep learning curve; or
- The need to adopt a new programming approach (e.g. noweb).[2]

[1] LyX users must however be familiar with the term "evil red text".

[2] A new approach may not be an entirely bad thing, but the resulting fragmentation may not be kind to those used to reading "more normal" code.

Why the name?

Conventional documentation of computer code often looks tacked on, resembling the stumpy, customarily docked tail of a large rottweiler. Dogwagger tries to address this problem by integrating the documentation and the program. The program becomes something which is pulled out of the documentation, rather than the other way around.

For example, in the great programming language Perl, there's a convention that allows you to mark sections of the program with an equals sign, followed by a name. All of the subsequent code is ignored by Perl until a magic line beginning with the expression `=cut` is encountered. A separate program can then be used to pull out the *cut* sections and assemble them into some sort of documentation. This approach is called POD, or 'Plain Old Documentation'. Dogwagger, although it is written in Perl, is more sophisticated, as it works the other way around. The tail wags the dog.

Why LaTeX & Perl?

LaTeX is a good way to produce documents that are both functional and fairly elegant, especially if you're working on a large project and need good technical documentation. It's the document preparation system of choice for mathematical and astrophysical journals, and has been tried and tested over decades. Perl is powerful and available on almost any platform you care to mention.

How do I run Dogwagger?

You will need Perl installed on your system (most UNIX and Linux systems will have it; with MS Windows, consider Strawberry Perl or ActivePerl).

Open up a console box (otherwise known as a 'terminal' or the 'command line', e.g. Konsole in Ubuntu, or a DOS box in MS Windows), change to the directory where you've put the file Dogwagger405.pl, and type in:

```
perl Dogwagger405.pl MYFILE.tex
```

... for Dogwagger to work its magic. Type the following from the command line:

```
perl Dogwagger405.pl --help
```

... for a full list of options, or simply read on.

1.1 Dogwagger: Advantages & disadvantages

The few disadvantages have already been mentioned. In more detail the advantages are:

- Program code and documentation are seamlessly integrated;
- Updates are concurrent. You can update program and documentation at one go, and generate program *or* documentation by simply submitting the same file to either PDFLaTeX or Dogwagger;
- Binary code can be integrated (as required) with other code, without needing fancy tools or many different types of file;
- As Dogwagger is available under the GNU public licence, it'll aways be freely available. You can download the source code of this file (Dogwagger405.lyx; Dogwagger405.tex) from GitHub: github.com/jvanschalkwyk.

1.2 GNU Public Licence

NB. This program is distributed under the Gnu Public Licence (GPL). A copy should accompany any distribution. For details of the GPL, see Appendix A, at the end of this document. This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

2 How to use Dogwagger

If you read the LaTeX source for *this file*, (available as both .lyx and .tex documents) then you'll see Dogwagger in action! But to get a bit more than a flavour read the following documentation thoroughly. The basic idea is:

1. Create your LaTeX documentation;
2. At will, insert sections of program within the documentation as `\verbatim` comments — these chunks will eventually all be concatenated into one program file!
3. Dress things up a little.

That's really it. The dressing up is very easy indeed. You've already encountered the most important piece of 'dressing up', the Dogwagger title line:

```
% Dogwagger version=‘4.0.5’ fileTarget=‘foo.pl’
```

If such a line isn't present, and you submit a file for Dogwagger to parse, then she will complain bitterly. You'll also get warnings if the version number is wrong. Note the backticks (‘) at the start of ‘quoted items’. The `fileTarget` instruction is self-explanatory.

There are a two other parameters you will almost certainly use on this important title line. Examples are:

- `startComment=‘#’`
- `noWarn=‘yes’`

The `startComment` option specifies how lines will be commented out. This is important because DogWagger normally writes a few lines at the start of a generated file saying where the file came from and when it was created. In addition, separate sections in the generated program code are separated by comments. The default comment character is #.

Different programming languages use different character sequences to signal a comment line — for example, C++ uses a double slash `//`, and Perl uses # But we can even implement the old-fashioned C `/*` comment style `*/` with the following combination of instructions:

```
% Dogwagger startComment=‘/*’ endComment=‘*/’ [etc]
```

By default, Dogwagger kindly warns you before it overwrites files, but you can override this behaviour using `noWarn=‘yes’` .

2.1 Suppress unwanted *verbatim* sections

If you don't want the next `verbatim` section to appear in code, say:

```
% Dogwagger dogsAllowed=‘no’
```

2.2 Make multiple files

Part of the way through your documentation, you may wish to terminate the current code (file) you're creating, and start a new file. For example, you may have discussed (and created) the main CPP program, and now wish to do the same for the .H header file. This is pretty easy. In the line *immediately preceding* the next `verbatim` section, insert a commented line similar to the following:

```
% Dogwagger newTarget=‘foobar.h’ startComment=‘//’
```

Remember to make the very first character of the line a '*%*', so as to comment the line out in LaTeX. See how we use newTarget to remind ourselves that this isn't the first file specified in the title line, but a subsequent one.

2.2.1 Files within directories

We deliberately don't encourage files generated by DogWagger to be written to obscure locations. Files can however be written to an *existing* subdirectory thus:

```
% Dogwagger newTarget=‘foo/bar.c’ startComment=‘//’
```

This statement creates the file *bar.c* in the subdirectory *foo*. Always use a forward slash (a la Unix) not the DOS/MS Windows backslash. Dogwagger will *not* create the subdirectory.

2.3 Deferred code

You can *defer* writing of a code section until other sections on which it *depends* have been written to the output file. For example, when discussing SQL code, we might wish to talk about the main table first, but in the final code we will first want to define the minor tables on which the main table depends!

To make use of this facility, use the following commands:

```
% Dogwagger dependsOn=‘alpha’
```

. . . or even, if something depends on several other sections:

```
% Dogwagger dependsOn=‘alpha,beta,gamma’
```

The name of the section depended on is then given by:

```
% Dogwagger myName=‘alpha’
```

When a dependsOn statement is encountered the assumption is made that *all* of the names depended on have *not yet been defined*! If any of them existed, then the smart user would simply leave them out!

An item can have entries for both myName and dependsOn. In this case, the name is kept pending until all dependsOn blocks have been created, at which point the item is written to output; then only is the name of the item itself resolved.[3]

It's important to realise that if two sections of code A and B are deferred pending the writing of code C, then the only way to ensure that B is written after A is to explicitly state that B depends on A.

2.4 Include binary files

Because binary files, especially executable ones, are inscrutable (and potentially harmful) you should generally avoid spreading them around, but occasionally it may be necessary to include such a file with your source code. Dogwagger meets this need by allowing UUencoded files to be included in `verbatim` sections thus:

```
% Dogwagger newTarget=‘uudecode’
```

In other words, the only 'filename' which is reserved is the case-sensitive uudecode. If this name is specified, then (as is usual for uunecoded files) the filename is picked out of the subsequent uuencoded information, located within the verbatim statement. There's a (uuencoded) uunecoding program in Appendix B.[4]

Because uuencoded text for a single file must be contained in one `verbatim` section, and this file is written immediately, you can write such a file from within the middle of creating other code. (This means that, in contrast to `newTarget`, you should not specify uuencode as the `fileTarget` parameter).

2.5 Minor frills and restrictions

In between the beginning and end of each 'verbatim' section there *must* be at least one line of data. If you put a carriage return immediately after the `\begin{verbatim}` it will be ignored. However, a carriage return just before an `\end{verbatim}` statement will be included in the output code. Play with this feature.

[3] Self-dependence will be unresolved, causing an error.

[4] In Ubuntu Linux, try "sudo apt-get install sharutils".

There are several little conveniences in DogWagger. You can label the comment at the start of each section using a line like the following *immediately preceding* a `verbatim` section:

```
% Dogwagger sectionTitle='This is a BIG BOLD section'
```

We can do slightly more fancy things:

```
% Dogwagger sectionTitle='Foo: Section $[SECTION]'
```

... which actually uses Dogwagger's internal section counter to replace `$[SECTION]` with the relevant section number.

2.5.1 Write code as a single line

Occasionally it's convenient to write several lines of verbatim text as a single line of output. Dogwagger to the rescue with the `oneLine='yes'` command!

Note that in this mode, trailing spaces count, but the leading spaces on the next line are removed. All other whitespace is preserved *as is*.

2.6 Make a debug version!

Here's a command which you can *only* include in the title line:

```
include='everything'
```

This wrinkle allows you to create two versions of code, a *debug* version, and a production version. By default, if you *omit* the above command from the title line, then the production version is created. If you include it, then we make a debug version. And what's the difference? Well, if Dogwagger encounters a line within a `verbatim` section which begins with the sequence:

```
+OPTIONAL
```

... then by default all of the code is *omitted* until:

```
-OPTIONAL
```

is encountered later on. 'Including everything' forces Dogwagger to include this optional code.

In many languages, there are ways of creating debug versions, for example the C++ `#define` followed by `#ifdef` and so on, but our way is more explicit and simpler.[5]

[5] We considered having an optional parameter after the OPTIONAL statement to allow multiple versions, but rejected this as extremely silly.

2.7 Use Dogwagger with LyX

I believe that LyX, the first "what you see is what you mean" word-processor, is now mature enough to supplant other LaTeX editors. Put `\begin{verbatim}` and `\end{verbatim}` statements and contained code within "Evil Red Text" boxes (ERT, press Control+L). There are only two catches:

1. ERT is raw LaTeX, so be careful to balance begin/end statements, braces, $ and so forth. (Also use `\verb` with great caution).

2. It's best to start the ERT with a comment line, before the actual % DogWagger line. This is because LyX may otherwise sometimes confuse Dogwagger by putting other characters before the % at the start of the % DogWagger line.

The very first Dogwagger line is best put in the LaTeX preamble (Document | Settings).

3 The source code of Dogwagger

Here's the Dogwagger version 4.0.5 code.

```
#!/usr/local/bin/perl -w
use strict;

  ######################################################################
  # This program is distributed under the Gnu Public Licence (GPL).
  # You should have received a copy of the GNU General Public License
  # along with this program; if not, write to the Free Software
  # Foundation, Inc., 59 Temple Place - Suite 330,
  # Boston, MA  02111-1307, USA.
  ######################################################################

my $ERRCOUNT=0;         # global. Ugh.
my $LINECOUNT=0;        # likewise
my $OPTN=0;             # global that supports +OPTIONAL
my $MAJORVERSION = 4;   # * REMEMBER TO CHANGE FROM VERSION 4.0.5
my $MINORVERSION = 0;   # * REMEMBER TO CHANGE FROM VERSION 4.0.5
my $TV  = 5;            # * version 4.0.5 (TV = tinyversion)
```

3.1 Initialisation

We get the current (system) time (Section 7.6), and then create several important arrays used for processing of deferred code.

```
my $TODAY = &GetLocalTime();
my @CHILDREN;
my @DEPENDENCIES;
my @PENDINGNAME;
my $DEFAULTNEWLINE = "\n";
my $NEWLINE = $DEFAULTNEWLINE;
```

The deferred arrays are CHILDREN (an array of sections which depend on other named sections, and haven't yet been written to file), DEPENDENCIES which stores the corresponding names *depended on*, and PENDINGNAME, which stores the name of each child, if that child has a name.

Each child contains multiple lines which will only be written to file when all of the dependencies of the child have been satisfied. As each name is encountered, it is removed from each dependency list where it occurs. If a dependency list becomes *empty* in this process, the corresponding item is written to output.

The variable `$NEWLINE` is a global that lets us specify the newline character(s) throughout all files that are printed.

3.2 Setup for file reentrancy

As of Version v4.0.4 (tme), this code initialises the associative array `%APPENDTOFILE` that lets the program know whether to append to a file or just overwrite it. This tiny amendment allows you to re-specify a file multiple times — each new invocation of fileTarget=... appends the subsequent code to the file!

```
my %APPENDTOFILE = (); # v4.0.4 (tme)
```

4 Startup

We handle the command line `--help` option with a help message:

```
my ($HelpMessage) = "\n      perl Dogwagger"
 . " $MAJORVERSION$MINORVERSION$TV.pl SOURCENAME.tex [log=WAGLOG.LOG]\n"
 . "         See: http://www.anaesthetist.com/mnm/dogwagger/\n"
 . "=================================================================\n"
 . "  *Picks out program code from within LaTeX verbatim statements*\n"
 . "       Within the .tex file, a DOGWAGGER LINE starts with \n"
 . "% Dogwagger \n"
 . "           ... followed on that line by one or more of:\n"
 . "   Option (example)                    Meaning\n"
 . " --------------------     ---------------------------------------\n"
 . " version='4.0.5'      ... state optimal Dogwagger version to use\n"
 . " fileTarget='FOO.BAR' ... the name of the very FIRST target file\n"
 . " [newLine='\\n\\r']     ... replace newline characters with these.\n"
 . "       <<ONLY use the above 3 in the first DOGWAGGER LINE>>\n"
 . " newTarget='NEW.BAR'  ... end off preceding file, start new file\n"
 . "     <<on starting a new file, all of the following options \n"
 . "       are reset, apart from startComment and endComment>>\n"
 . " startFile='<?php\\n'  ... very first text in file, default is ''\n"
 . " endFile='\\n?>'       ... final text in file, default is also ''\n"
 . " noHeader='yes'       ... Dogwagger will now  not write a header\n"
 . " noTail='yes'         ... omit all Dogwagger code at end of file\n"
 . " startComment='<!--'  ... define how a Dogwagger comment starts;\n"
 . " endComment='-->'     ... how comment ends, (default is nothing)\n"
 . " noWarn='yes'         ... do NOT warn before overwriting a file.\n"
 . "    <<the above 7 can only be used on starting a new file>>\n"
 . " dogsAllowed='no'     ... do NOT include following verbatim text\n"
 . " sectionTitle='BLAH'  ... force use of this as new section title\n"
 . " noSections='yes'     ... turn section titles on  (or off: 'no')\n"
 . " oneLine='yes'        ... in this block alone, concatenate lines\n"
 . "                          (with suppression of leading spaces)\n"
 . " dependsOn='ALPHA'    ... defer writing text  until name defined\n"
 . " dependsOn='A,B,C'    ... defer  until multiple blocks are named\n"
 . " myName='ALPHA'       ... name a block  (this is case sensitive)\n"
 . " newTarget='uudecode' ... just extract a single uuencoded file.\n"
 . "    <<See the documentation for +OPTIONAL code inclusions>>\n"
 . " *-------------------------------------------------------------*\n"
 . " Your LaTeX code must include, in its first 40 lines, a DOGWAGGER\n"
 . " LINE that describes  both the version and the first target file.\n"
 . " All subsequent LaTeX verbatim sections are appended, by default.\n"
 . " To modify behaviour, put a DOGWAGGER LINE immediately preceding\n"
 . " the first line of a verbatim statement, using the above options.\n"
 . " *-------------------------------------------------------------*\n"
 . "   Dogwagger 4.0.5 is Copyright (C) J van Schalkwyk, 2005--2020\n"
 . "      Made available under the GNU General Public Licence v2.\n"
 . "=================================================================\n";
```

After printing a Dogwagger banner (and the date), we obtain the name of the .tex source file from the command line (ARGV). If this is instead a request for help, we print the help message and exit; otherwise we validate the file name.

```
my ($fido) = "";  # file name
print "\n\n                      ======================\n"
        .    "                        Dogwagger Version "
        .                  "$MAJORVERSION.$MINORVERSION.$TV \n"
        . "                      ======================\n";
print ("                       $TODAY\n");
$fido = $ARGV[0]; # command line
if ( $fido =~ /--help/i )
   { print $HelpMessage;
     exit;
   } else
   { print "                        (Try --help for help)\n"; };
if ( !(defined $fido) || ((length $fido) < 5))
   {  die "Error. Bad file name. "
         . "Please submit a valid name e.g. foo.tex";  };
```

We check for a log file (a second, optional, command-line argument **log=FOO.log**), and otherwise default to the name WAGLOG.LOG. We open this log, and then invoke WagTheDog:

```
my $filelog = $ARGV[1];   # format is: log=filename.xxx
if ($filelog =~ /log=(\w+\.\w+)/i )
   { $filelog = $1;       # pull out name
   } else
   { $filelog="WAGLOG.LOG";
   };
open FILELOG, ">$filelog" or
     die "*CRASH* Could not open LOG <$filelog> :$!\n";
print FILELOG "LaTeX DogWagger, Version "
      . "$MAJORVERSION.$MINORVERSION.$TV [$TODAY]";
&Caution ("INPUT file: <$fido>");
my $cSECTIONS= 0;         # 4 globals: count verbatim sections,
my $cFILES   = 1;         #   generated files (must be >=1),
my $cSKIP = 0;            #   sections skipped,
my $cOUST = 0;            #   and files overwritten.
my $ERRCODE = 0;
if ( (my $fail = WagTheDog($fido))
   ||($ERRCOUNT > 0)
   ) { print "\n*Problems: $fail, Errors=$ERRCOUNT\n";
       $ERRCODE = 99;
     };
$TODAY = &GetLocalTime();
&Caution( "Done [$TODAY]. Sections:$cSECTIONS($cSKIP skipped), "
      . "files:$cFILES ($cOUST overwritten).\n\n" );
exit($ERRCODE);
```

The main function WagTheDog (§5) returns an error message if it fails. A global error count over zero also indicates failure.

5 The principal function

As the name suggests, WagTheDog does the work. WagTheDog itself has several ugly features[6] — let's see how DogWagger copes!

```
sub WagTheDog
{ my($fido);
  ($fido)=@_;
  my(@CHILDREN, @DEPENDENCIES, @PENDINGNAME);
  my $RETAINTARGETLINE = ''; # used for new file parameters
```

5.1 Preliminaries

After some debugging statements and a check for the presence of a filename string, we clear the various arrays, and set up `myNam`, which stores the name of the current block of code that is being written. Once we've finished writing this block, we will resolve all of the dependencies on `myNam`.

```
  @CHILDREN = ();
  @DEPENDENCIES = ();
  @PENDINGNAME = ();
    $CHILDREN[0]='';
    $DEPENDENCIES[0]='';
    $PENDINGNAME[0]='';
  my($myNam);
```

Let's open the source file, failing if this opening fails:

```
  $LINECOUNT = 0;
  my ($E1) = 0;
    open FIDO, $fido
         or $E1 = &GlobalError("Could not open source <$fido> :$!");
if ($E1)
   { return ($E1);
   };
```

[6] Some of these are a legacy from when DogWagger had a GUI, others are just bad programming.

5.2 Find the header line

Next, scan through the first n=100 lines for the header line:

```
  my($i) = 100;
while ($i > 0)
   { $_ = <FIDO>;
     $LINECOUNT ++;
   if ( /\%.* Dogwagger/i )
      { $i = 0; }; # force end
        $i --;
   };
if (! $i)  # if DogWagger found, $i should be -1.
   { &Caution("DogWagger data not found in <$fido>");
      close FIDO;
      return ('No data');
   };
  my($hotline) = $_;  # redundant
  $NEWLINE = &GetNewLine($hotline); # determine newline code
```

We fail if the first Dogwagger line is not encountered. The only requirements are the presence of the term ' Dogwagger', and that the line is commented out à la LaTeX.

5.3 Check the version

We will soon check for version compatibility, and extract Dogwagger command data into the variables defined below:

```
  my ($version, $DOGFILE, $startComment, $nowarn, $sft,
      $eft, $endComment, $nohead, $nosections, $notail);
  # amended 2011-12-18
  my($majorVersion, $minorVersion);
```

All variables apart from those pertaining to comments are reset to default (null string) after the closure of the current file, so for each file with starting and/or ending text, the values must be specified anew! ReadHeader (Section 6) obtains the values:

```
  my($MANDATORY);
```

```
  ($version, $DOGFILE, $startComment, $nowarn, $MANDATORY,
   $sft, $eft, $endComment, $nohead,
   $nosections, $notail) = &ReadHeader($hotline);
   # nohead.. added 2011-12-18

if ($version == 0) # error
   { return ('Bad header');
   };

  $_ = $version;
  /(.+)\.(.+)\.(.+)/;  # pull out major and minor version numbers:
  $majorVersion = $1;
  $minorVersion = $2;  # ignore trivial version number = $3
```

We check the version specified against the current version of Dogwagger:

```
if ($majorVersion > $MAJORVERSION)
   { &Caution(
     "Warning: DogWag(V$MAJORVERSION.$MINORVERSION) "
    ."won't support all features of V$majorVersion.$minorVersion");
   } else
   {
   if (  ($majorVersion == $MAJORVERSION)
       &&($minorVersion > $MINORVERSION)
      )
      { &Caution( "Caution: minor version switch. "
                 ."Problems may abound!");
      };
   };
```

Dogwagger emits appropriate warnings if either the major or minor version numbers of Dogwagger are incompatible with those in the file being translated. Trivial version numbers (the third part of the dotted version number) are ignored.

5.4 Open the target file

We open the target file (Section 9.1), using the name provided, and fail if this fails.

```
  my ($ok, $wagline);
if (! OpenTargetFile($DOGFILE, $startComment, $fido,
                     $nowarn, $sft, $endComment, $nohead))
```

```
                    # nohead.. added 2011-12-18
   { return ("Could not open target <$DOGFILE>");
   };  #fail
```

5.5 Read source

Now we're ready to read in the source file, and process it. Several startup flags control interpretation, the most important being `ishot`, which determines whether we are actively writing lines, or just throwing away LaTeX text.

```
  my($ishot, $hotdata, $chomper, $chomped, $hotline);
  my($nodogs);
  my($SECTION) = 1;
  my ($SECTIONTITLE) = ''; # default is empty

  $ishot   = 0;
  $chomper = 0;  # default is OFF
  $chomped = 0;
  $nodogs  = 0; # default
```

5.6 An enormous *while* statement

A biig `while` statement surrounds everything, within which we read each line in turn and process it.

```
while (1)            #         an enormous while statement ****
   { $_ = <FIDO>;
   if (! defined)    # exit.
      { close FIDO;
        print FILELOG "\n  Line $LINECOUNT: "; # preliminary to closing
        &CloseDogFile($startComment, $eft, $endComment, $notail);
        return(0);
      };

     $_ = &ChompLine($_);
     $LINECOUNT ++;
   if (! $ishot) # if not writing
```

In the above, ChompLine (Section 7.3) chomps the terminal 'newline' character(s) off the input line that is read from the file handle `FIDO`.

5.7 Not hot

The following code deals with the case where we are not 'hot', i.e. not writing to a target file.

```
{
if ( /(.*)\\begin\{verbatim\}(.*)/ ) # if "begin verbatim":
   { $cSECTIONS ++;    # bump verbatim section count
   if (! $nodogs)
      { $hotdata = $2;   # amended 2011-12-19
      if ($1 !~ /\%/ ) # if verbatim not commented out
         { $ishot = 1; # turn on
           $SECTION = &PrintSectionHeader($startComment,
              $SECTION, $endComment, $nosections,
              $SECTIONTITLE);
           print DOGFILE $hotdata;   # clumsy but explicit
         };
      };
```

The above deals with the case where we've just encountered `\begin{verbatim}`. We ensure that this statement hasn't been commented out in LaTeX, and if not, print a section header (See §10.1). Otherwise, we check for a Dogwagger instruction:

```
} else                                  # NOT "begin verbatim":
{ my($depOn);
  $myNam = '';
  $depOn = '';
  $nodogs = 0;

if (/^\s*\%\s*DogWagger/i)              # if IS DogWagger ***
   { &PrintLogLine ( "{wag} ");
   if ( /dogsAllowed=\'no\'/)           # dogs NOT allowed
      { $nodogs = 1;
        $cSKIP ++;
        print FILELOG "__skip__";
      } else                            # dogs are allowed
```

If the instruction is "no dogs allowed" we set this flag; otherwise we look for the other common Dogwagger commands as follows.

```
      { $wagline = $_;              # dogs are allowed **
      if ( ! /^(.*)newTarget=\'(.+?)\'(.*)$/ ) # revised 4.0.3
```

```
{
if (/^(.*)dependsOn=\‘(.+?)\’(.*)$/)
   { $depOn = $2;
     $_ = "$1$3";
     print FILELOG "dependencies <$depOn>; ";
   };

if (/^(.*)myName=\‘(.+?)\’(.*)$/)
   { $myNam = $2;
     $_ = "$1$3";
     print FILELOG "name=$myNam; ";
   };

if (/^(.*)oneLine=\‘yes\’(.*)$/)
   { $chomper = 1;        # turn on!
     $_ = "$1$2";
     print FILELOG "(chomp) ";
   };

if (/^(.*)sectionTitle=\‘(.+?)\’(.*)$/)
   { $SECTIONTITLE = $2;
     $_ = "$1$3";
     print FILELOG "title<$SECTIONTITLE> ";
   };

if ( /^(.*)noSections=\‘(.+?)\’(.*)$/ )
   { print FILELOG "nosections=$2 ";
   if ($2 eq ’yes’)
      { $nosections=1;
        $_ = "$1$3";
      }
   elsif ($2 eq ’no’)
      { $nosections=0;
        $_ = "$1$3";
      };             # if neither yes nor no, ignore!
   };
if ( /(\w+\s*=\s*\‘.+\’)/ )  # implies unknown command
   { &Caution("*Warning* LINE $LINECOUNT "
            . "Unknown/duplicated "
            . "Dogwagger command(s) <$1>");
   };
```

The noSections test allows you to turn section headers on or off at will, so we need to accommodate not just the 'yes' option but also the 'no' option. Specifying something other than 'yes' or 'no' will force an error, as $_ is then not updated, and Section **??** will catch this below. Although turning on noSections (disabling printing of section headers) is logically incompatible with sectionTitle, we don't check for this minor conflict.

5.7.1 A new file

The final legal Dogwagger instruction is newTarget. The code is here made a bit more complex because I need to accommodate the possibility that newTarget is a uuencoded file. (I've decreased the indentation, a reflection of the baroque complexity of WagTheDog).

```
                } else # IS newTarget
                {
##################deeply indented section#######################
$RETAINTARGETLINE = "$1$3";      # keep all other specifications
$DOGFILE = $2;                   # get name of new file
# the following line bumps file count _unless_ previously opened
$cFILES++ unless exists $APPENDTOFILE{ $DOGFILE }; # v4.0.4 (tme)
if ($DOGFILE =~ /^uudecode$/)
     # if uudecoding do NOT terminate current file!
   { my ($ufile, $umode, $uout) = Uudecode();
     if (length $ufile > 0)
        { &PrintLogLine ("uudecoding <$ufile> mode $umode");
          my ($E2) = 0;
          if (-e $ufile) { $cOUST++; }; # bump overwrite count!
          open UFILE, ">$ufile"
               or $E2 = &GlobalError("Uudecode failed <$ufile>");
          if (! $E2)
             { binmode UFILE;     # NB otherwise MSDOS stuffup!
               print UFILE $uout; # IGNORE UNIX mode in $umode.
               close UFILE;
        };
   };
```

The above code handles the case where the target name is precisely 'uudecode', decoding and writing the uuencoded file (and increasing the overwrite count $cOUST if the file exists). Otherwise close the current target file (Section 9.2), read new target parameters (ReadTargetParams, Section 6.1), and open a new file (See Section 9.1).

```
    } else                          # close current, open new!
    { &CloseDogFile($startComment, $eft, $endComment, $notail);
      # close with old parameters; next, read new...
      ($startComment, $nowarn, $MANDATORY, $sft, $eft,
       $endComment, $nohead, $nosections, $notail) =
      &ReadTargetParams($RETAINTARGETLINE, $startComment,
                        $endComment);
      print FILELOG ("\n  Comment format now \"$startComment"
                     . "foo$endComment\"");
      if (! OpenTargetFile($DOGFILE, $startComment, $fido,
            $nowarn, $sft, $endComment, $nohead))
              # nohead.. added 2011-12-18
         { return ("Could not open target: <$DOGFILE>");  #fail
         };
    };
##################end deeply indented section###################

                  }; # END of else (is newTarget)
               };    # end of "dogs are allowed"
            };         # end of IS DogWagger ***
```

Finally, if there are dependencies, we store these (StoreChild, Section 8.1).

```
         if (length $depOn > 0)  # if dependency
            {
            if (! &StoreChild ($myNam, $depOn, $chomper)) # keep whole
               { &Caution("WARNING: \
                 Input file <$fido> terminated unexpectedly!");
                 close FIDO;
                 close DOGFILE;
                 return ('Sudden INPUT failure'); #fail!
               };
              $myNam = ''; # cannot YET resolve (is a dependency).
            };
         };  # end else ... not begin verbatim.
```

5.8 Hot code

The preceding code dealt with the case where were are not 'hot'. If we *are* busy writing lines to output (are hot) we next check for the end of a `verbatim` statement.

```
} else # ARE HOT (ARE WRITING):
{
if ( /(.*)\\end\{verbatim\}/ ) # end verbatim?
   {
   if ($OPTN) # OPTION still on?
      { &Alert ( "Optional text not closed. See log!");
        &GlobalError("\n ERROR line $LINECOUNT: "
                    . "+OPTIONAL not closed");
        $OPTN = 0;
      };

     $hotdata = $1;
     print DOGFILE $hotdata; # last chunk.
     print FILELOG " ... $LINECOUNT.";
   if (length $myNam > 0)  # if name defined
      { $SECTION = &FixName($myNam, $startComment,
                            $SECTION, $endComment,
                            $nosections, $SECTIONTITLE);
      };
     $ishot = 0;             # turn off.
     $chomper = 0;           # back to default
     $chomped = 0;           # redundant.
     $SECTIONTITLE = '';
   } else  # NOT end verbatim..
```

If it is the end of a verbatim statement, we print the last bit of text just before the end. Then, if `myNam` contains a name, we resolve all of the dependencies on that name using the function FixName (Section 8.2).

Otherwise (it's not the end of a verbatim statement) we check for the case where we are 'chomping' lines (the oneLine option forces multiple verbatim lines to be concatenated, removing all newline characters).

```
   {
   if ($chomped)           # Already chomped?
      { / *(.*)/;          # remove leading spaces
        $_ = $1;           # (even allow null line)
```

```
      };
   if ($chomper)           # IF line must be chomped
      { print FILELOG '+'; # Amended 2011-12-18
        $chomped = 1;      # signal we've just chomped.
      } else               # UNLESS chomping,
      { $_ = "$_$NEWLINE"; # restore a default newline!
      };
```

We also check for +OPTIONAL code, only written if we specified `include='everything'`

```
   if ( /^\s*\+OPTIONAL(.*)$/)
      { $OPTN = 1;
        $_ = "";
      if (length $1 > 0)
         { &Caution("Warning LINE $LINECOUNT. "
                  . "Extra +OPTIONAL text");
         };
      };
   if ( /^\s*-OPTIONAL(.+)$/)
      { $OPTN = 0;
        $_ = "";
      if (length $1 > 0)
         { &Caution("Warning LINE $LINECOUNT. "
                  . "Extra-OPTIONAL text");
         };
      };

   if ($MANDATORY || ! $OPTN) # unless optional is active
      { print DOGFILE $_; # write to output
      };

   }; # end else not end verbatim
};   # end else are hot
```

5.9 Finish off

We finish off our enormous `while` statement, and close the source file.

```
  }; # end of enormous while stmt.

}  # end of WagTheDog
```

6 Reading the header

As discussed in the introductory section, we must accommodate the various header line options. We obtain the version, fileTarget and newLine values in the following routine:

```
sub ReadHeader
{ my ($hotline);
     ($hotline) = @_;
  my ($ver, $target, $comment, $nowarn, $mandatory,
      $sft, $eft, $endComment,
      $nohead, $nosections, $notail);      # $nohead 2011-12-18
  $ver = 0;
  $target = '';
  $comment = '#';                          # aka startComment
  $nowarn = 0;
  $mandatory = 0;
  $sft = '';
  $eft = '';
  $endComment = '';
  $nohead = 0;                              # default write head
  $nosections = 0;                          # and section
  $notail = 0;
  if ( $hotline !~ /^(.*)version=\'(\d+\.\d+\.\d+)\'(.*)$/ )
     { &Caution ("Missing/defective version number");
       return (0,0,0,0, 0,0,0,0, 0,0,0);    #fail
     };
  $ver = $2;
  $hotline = "$1$3";
  if ( $hotline !~ /^(.*)fileTarget=\'(.+?)\'(.*)$/ )
     { &Caution ("Missing/defective target filename");
       return (0,0,0,0, 0,0,0,0, 0,0,0);    #fail
     };
  $target = $2;
  $hotline = "$1$3";
  if ($hotline =~ /^(.*)newLine=\'(.+?)\'(.*)$/ )
     { $NEWLINE = $2;
       $hotline = "$1$3";
       $NEWLINE =~ s/\\n/\n/g;              #
       $NEWLINE =~ s/\\r/\r/g;              # replace \n =r
     };
```

```
  ($comment, $nowarn, $mandatory, $sft, $eft,
   $endComment, $nohead, $nosections, $notail) =
       &ReadTargetParams($hotline, $comment, $endComment);
  return ($ver, $target, $comment, $nowarn, $mandatory,
          $sft, $eft, $endComment,
          $nohead, $nosections, $notail);  }
```

Other parameters are obtained by ReadTargetParams (§6.1).

6.1 Read target file parameters

We read in parameters that are directly related to file creation. These parameters do *not* include the version number and the fileTarget parameter. In version 2.1 we added the option to specify the very first few characters of the file using the *startFile* option. This is useful for HTML and PHP. See also the corresponding endFile option for terminating the last chunk of a file.

```
sub ReadTargetParams
{ my ($topline, $comment, $endComment);
     ($topline, $comment, $endComment) = @_;
  my ($nowarn, $mandatory, $sft, $eft,
      $nohead, $nosections, $notail);  # $nohead.. added 2011-12-18
  my ($alterSC, $alterEC);
  $nowarn    = 0;
  $mandatory = 0;
  $sft       = '';
  $eft       = '';
  $nohead    = 0;           # default write head
  $nosections= 0;           # and section
  $notail    = 0;
  $alterSC   = 0;           # ON if alter startComment
  $alterEC   = 0;           # on if alter endComment

  if ($topline =~ /^(.*)include=\‘everything\’(.*)$/)
     { $mandatory = 1;
       $topline = "$1$2";
       print FILELOG "include ALL "; };

  if ($topline =~ /^(.*)startComment=\‘(.+?)\’(.*)$/)
     { $comment = $2;       # new startComment
       $topline = "$1$3";
```

```
      print FILELOG "comment '$comment'";
      $alterSC = 1;
   };
if ($topline =~ /^(.*)newComment=\‘(.+?)\’(.*)$/) # obsolete.
   { &Caution(  "Minor warning: newComment is deprecated "
              . "(LINE $LINECOUNT). Use startComment." );
      $comment = $2;
      $topline = "$1$3";
      print FILELOG "comment '$comment'";
      $alterSC = 1;
   };
if ($topline =~ /^(.*)endComment=\‘(.*?)\’(.*)$/)
   { $endComment = $2;
      $topline = "$1$3";
      print FILELOG "endComment '$endComment'";
      $alterEC = 1;
   };
```

Because the comment format is carried over, there's a potential problem if an old format has an endComment (e.g. `-->`) and we change the startComment alone. We thus *reset* endComment if we've altered startComment but not endComment:

```
if ($alterSC && ! $alterEC) # if only alter startComment
   { $endComment = '';      # force endComment to default.
   };
```

The remaining options are straightforward:

```
if ($topline =~ /^(.*)noWarn=\‘yes\’(.*)$/)
   { $nowarn = 1;
      $topline = "$1$2";
      print FILELOG "warn is OFF; "  };
if ($topline =~ /^(.*)startFile=\‘(.*?)\’(.*)$/) # ver 2.1
   { $sft = $2;
      $topline = "$1$3";
      print FILELOG "start code {$sft}; ";
      $sft =~ s/\\n/$NEWLINE/mg; # CR's !!
   };
if ($topline =~ /^(.*)endFile=\‘(.*?)\’(.*)$/)   # ver 2.1
   { $eft = $2;
```

```
      $topline = "$1$3";
      print FILELOG "end code {$eft}; ";
      $eft =~ s/\\n/$NEWLINE/mg; # CR's !!
   };
if ($topline =~ /^(.*)noHeader=\'yes\'(.*)$/ )   # 2011-12-18
   { $nohead=1;
     $topline = "$1$2";
     print FILELOG "NO header; " };
if ($topline =~ /^(.*)noSections=\'yes\'(.*)$/ )
   { $nosections=1;
     $topline = "$1$2";
     print FILELOG "NO sections; "  };
if ($topline =~ /^(.*)noTail=\'yes\'(.*)$/ )
   { $notail=1;
     $topline = "$1$2";
     print FILELOG "NO tail; "  };

if ($topline =~ /(\w+\s*=\s*\'.+\')/ )  # v 4.0.3
   { &Caution("*Warning* LINE $LINECOUNT Unknown/duplicated "
              . "Dogwagger NEWFILE command(s) <$1>");
   };

return ($comment, $nowarn, $mandatory, $sft, $eft,
        $endComment, $nohead, $nosections, $notail);  }
```

7 Miscellaneous routines

The following are trivial:

7.1 Confirm an action

Get console input (confirmation, i.e. ‘y’):

```
sub Confirm
{ my ($msg);
     ($msg) = @_;

     print "\n$msg";
     my($ans);
     $ans = <STDIN>; # get stdin
     if ($ans =~ /^y/i )
        { return(1);
        };
    return (0);  }
```

7.2 Alert

Alert the user with a warning. Do not alter $_.

```
sub Alert
{ my ($msg);
     ($msg) = @_;
     print "$msg";  }
```

7.3 ChompLine

Remove CR/LF characters from the start/end of a line:

```
sub ChompLine
{ my ($line);
     ($line) = @_;
  if ( $line =~ /([^\n\r]*)/ms )
                { $line = $1;
                };
  return($line);  }
```

7.4 Determine newline character(s)

Given a sample line, find out the codes used eg 0D or 0D0A. Unescape is below (§7.8).

```
sub GetNewLine
{ my ($line);
     ($line) = @_;
  if ( $line =~ /[^\n\r]*([\n\r]+)$/ms )
                { my($nl) = $1;
                  $_ = &Unescape($1);
                  print FILELOG "\n  newline is <$_> ";
                  return($nl);
                };
  print FILELOG "\n  Resorting to newline default.";
  return($DEFAULTNEWLINE); }
```

7.5 Caution — Alert with print

'Caution' always generates an alert. This code should not alter $_.

```
sub Caution
{ my ($msg);
     ($msg) = @_;
  print FILELOG "\n$msg";  # must prepend newline.
  &Alert("\n$msg"); }
```

7.6 Read the time

This is the local machine time.

```
sub GetLocalTime
{  my ($sec, $min, $hour, $mday, $mon,
       $year, $wday, $yday, $isdst);
($sec, $min, $hour, $mday, $mon,
       $year, $wday, $yday, $isdst) = localtime(time);

  $year += 1900;      #fix y2k.
  $mon ++;            #january is zero!
  return ("$year-$mon-$mday $hour:$min:$sec"); }
```

7.7 An error-related hack

Clumsy.

```
sub GlobalError
{ my ($msg);
     ($msg) = @_;
  print FILELOG "\n$msg";
  $ERRCOUNT ++; # bump error count
  return($msg);
}
```

7.8 Pretty up codes

We look for \r and \n (carriage return, line feed) and substitute text representations:

```
sub Unescape
{ my ($t);
     ($t) = @_;
  $t =~ s/\n/\\n/ms;
  $t =~ s/\r/\\r/ms;
  return($t);
}
```

8 Deferred code

The following routines allow you to write code that is deferred until the code it depends on has been resolved. Do not assume that if A and B both depend on C, that A or B will be resolved in any particular order, unless you explicitly state that A depends on B, or vice versa!

8.1 Store array of lines

Here we keep 'child' (dependent) lines in an array, to be resolved later when all dependencies are met. The index of the topmost child is given by `$#CHILDREN`. The list of dependencies contains elements separated by commas, and there are starting and terminal commas to facilitate matching.

```
sub StoreChild
{  my ($pendingName, $dependencies, $chomper);
      ($pendingName, $dependencies, $chomper) = @_;
   my ($idx, $child);
   $idx = 1+$#CHILDREN;
   print FILELOG "Keeping child[$idx].";

   $_ = <FIDO>; # first line *must* be begin verbatim
                # what if this is not defined?? (check me)
   $LINECOUNT ++;
   $_ = &ChompLine($_);
   if ($chomper)           # IF line must be chomped
      { print FILELOG '/'; #
      } else               # UNLESS chomping,
      { $_ = "$_$NEWLINE"; # restore a default newline!
      };
```

The above allows a oneLine='true' statement to affect a deferred block. We now make sure that the line begins with a new `verbatim` statement.

```
   if ( /\\begin\{verbatim\}(.*)/ )
      { $child = $1;  # keep rest of line
      } else
      { print FILELOG "\n*ERROR at line $LINECOUNT:"
         . " no verbatim stmt on 1st child line!";
        $ERRCOUNT ++; # bump error
        print FILELOG "<$ERRCOUNT!>";
```

```
      print FILELOG "<$_>";
      return 1; # not fatal, per se.
    };
```

Next, store the name of the pending section, and its dependencies, all at the top of the current (global) lists. The actual text is stored in `$child`. See how the dependencies list has a leading comma.

```
$DEPENDENCIES[$idx] = ",$dependencies,";
$PENDINGNAME[$idx] = $pendingName;
$CHILDREN[$idx] = ''; # default nothing
my($ok) =1;
```

We sequentially read in lines until end-of-file (an error), or the verbatim statement ends.

```
while($ok)
  { $_ = <FIDO>;
    if (! defined)
       { return 0; # fail
       } else
       { $_ = &ChompLine($_);
         $LINECOUNT ++;
         if ($chomper)             # IF line must be chomped
            { print FILELOG '/';
            } else                 # UNLESS chomping,
            { $_ = "$_$NEWLINE";   # restore a default newline!
            };
        if ( /(.*)\\end\{verbatim\}/ )
           { $ok = 0;
             $child = "$child$1";  # append last section
           } else
           { $child = "$child$_";  # concatenate.
           };
       };
  };
$CHILDREN[$idx] = $child; # store away lines to be printed
return 1;
} # success!
```

8.2 Resolve labelling dependencies

FixName walks through all dependencies, resolves them (where relevant), and on resolution, writes the relevant child code to output. (A 'child' is a section which depends on other sections, and must not be written before these sections have been identified and written).

The arguments submitted are the name to be resolved, the codes for starting and ending a comment (c, ec), whether section headers will be written (nosections), and the `$SECTION` number. Note that the last-mentioned will be amended and returned.

```
sub FixName
{ my ($fname, $c, $SECTION, $ec, $nosections, $SECTIONTITLE);
     ($fname, $c, $SECTION, $ec, $nosections, $SECTIONTITLE) = @_;
     # nosections added 2011-12-18
  my ($morenames) = ",$fname,";
  my ($idx);
```

We now enter a while loop that takes the 'name to be resolved' off `morenames` and then enters an inner *while* loop that examines each pending name to see whether it's dependent on the name to be resolved.

```
  while ( $morenames =~ /^(.*,)(.+),$/ ) # split off last name
    { $fname = $2;
      $morenames = $1;
      $idx = $#CHILDREN;
      while ($idx > -1 )                 # for each child entry
        { if ($DEPENDENCIES[$idx] =~ /(.*,)$fname,(.*)/ )
             { $_ = "$1$2";              # if name in list, clip
               print FILELOG " (child[$idx] now has <$fname>) ";
               $DEPENDENCIES[$idx] = $_; # rewrite
               if ( /^,$/ )              # if all resolved
                  {  print FILELOG "WRITING child[$idx] ";
                     $SECTION = &PrintSectionHeader($c, $SECTION,
                                        $ec, $nosections,
                                        $SECTIONTITLE);
                     print DOGFILE $CHILDREN[$idx];
                     $CHILDREN[$idx] = '';
```

Once we've resolved this dependency, we realise that other files may in turn be dependent on the child that has been resolved. We get the name of *this* child off `@PENDINGNAME` and add it to `$morenames`.

```
                    # ....WAIT! this child may have dependencies!
                    if (length $PENDINGNAME[$idx] > 0) # if so ...
                       { $morenames =
                            "$morenames$PENDINGNAME[$idx],";
                            # add name of resolved child!
                         $PENDINGNAME[$idx] = '';# clear me!
            };   };    };
          $idx --;                      # move to preceding child
        };                              # all children done.
    };
  return $SECTION; }
```

The above explains why we *must* work backwards through `@DEPENDENCIES`. Earlier entries may depend on later ones, but later ones will have been resolved at entry. It's therefore wrong to start `$idx` at zero and count up.

8.3 Check for unresolved dependencies

At the end, we have to make sure that all dependencies have been resolved, or signal an error. Errors are also written to the log.

```
sub CheckUnresolved
{ my($idx);
  $idx = $#CHILDREN;
  my ($errcnt);
  $errcnt = 0;
  while ($idx > -1)
    {
      if (length $CHILDREN[$idx] > 0)
         { print FILELOG "\n\n *** ERROR *** "
                         . "\n\n Unresolved code: \n ";
           print FILELOG "Dependencies: <$DEPENDENCIES[$idx]> \n";
           print FILELOG "Name: <$PENDINGNAME[$idx]> \n"
                         . " <Code: <---\n ";
           print FILELOG $CHILDREN[$idx];
           print FILELOG "$\n ---> Code ends> \n\n";
           $errcnt++;
         };
      $idx --;
    };
  return $errcnt; } # number of errors, 0=ok.
```

9 Handle multiple files

This section is a consequence of Dogwagger's ability to generate multiple files from a single .TEX source. We chose the simple option of closing the first file and then opening and writing the next one, rather than having multiple dangling file handles. The sole exception to this rule is that if we are writing a UUdecoded binary file, we don't fiddle with the currently open file.

9.1 Open the target

We open a target file, warning appropriately about overwrites.

```
sub OpenTargetFile
{ my ($DOGFILE, $c, $fido, $nowarn, $sft, $ec,
      $nohead);  # nohead added 2011-12-18
     ($DOGFILE, $c, $fido, $nowarn, $sft, $ec,
      $nohead) = @_;
     my($ok);
  $TODAY = &GetLocalTime();
```

The following code sets the prefix to overwrite if the filepath does not occur in the hash \verb+$APPENDTOFILE+, otherwise to append.

```
  my $prefix = ">";  #  start amendment for v4.0.4 (tme)
  $prefix = ">>" if exists $APPENDTOFILE{ $DOGFILE };
  $APPENDTOFILE{ $DOGFILE } = 1;  # end amendment v4.0.4 (tme)
```

We check that the file exists, and warn if overwriting, unless this warning has been suppressed:

```
  if ($prefix eq ">" and -e $DOGFILE) # if prefix condition added v4.0.4 (tme)
     { if (! $nowarn)  # and warning enabled
          { if (! &Confirm (
                      "Overwrite <$DOGFILE>? Are you sure?"))
               { &Caution( "[NOT overwriting $DOGFILE]");
                 $DOGFILE = 'JUNK.JUNK'; # write to junk file.
               } else
               { print FILELOG "[Overwriting $DOGFILE]";
                 $cOUST++;  # increase overwrite count.
               };
           } else
```

```
          { $cOUST++;  # increase overwrite count.
    };    };
  my ($E3) = 0;
```

If not overwriting, rather than simply discarding the text, we rather clumsily write to the file *JUNK.JUNK*.

```
  open DOGFILE, $prefix."$DOGFILE" or  # ">" changed to $prefix v4.0.4 (tme)
        $E3 = &GlobalError(
                "Could not open target <$DOGFILE> :$!");
     if ($E3)
        { return 0;
        }; #fail
  print FILELOG "\n\n==Opened target file <$DOGFILE> ";
```

On opening the target file, we write several lines to this file (`DOGFILE`) using the comment character(s) to rem out the lines.

```
  print DOGFILE $sft; # very first text eg. for PHP.
                      # print EVEN IF noHead='yes'.
  if (! $nohead)      # added 2011-12-18
  { print DOGFILE
      "$c Generated by LaTeX DogWagger Version " .
      "$MAJORVERSION.$MINORVERSION.$TV "
      . "from file <$fido>$ec$NEWLINE";
    print DOGFILE "$c Date: [$TODAY] $ec$NEWLINE";
    print DOGFILE "$c Do NOT edit this file. "
            . "Edit the LaTeX source!!$ec$NEWLINE";
                      # \n appended 2011-12-18. Is this wise?
  };
  return 1; # success
}
```

The noHead='yes' option allows us to suppress the Dogwagger header text, but if we specified startFile text, it is still (of course) written.

9.2 Close target file

Simply close the output file handle `DOGFILE`.

```
sub CloseDogFile
{ my ($c, $eft, $ec, $notail);
     ($c, $eft, $ec, $notail) = @_;
  my ($EC) = &CheckUnresolved();
  $ERRCOUNT += $EC;      # global
  if ($EC > 0)
     { print FILELOG "\nUnresolved errors <$EC>";
       if (! $notail)
          { print DOGFILE
              "$NEWLINE$NEWLINE$c -- WARNING:"
              . " $EC errors. See log!$ec$NEWLINE";
     };   };
```

We can suppress writing of terminal (‘tail’) Dogwagger code, but the final code in `$eft` is written (if it exists) regardless of the value in `$notail`.

```
  if (! $notail)
     { print DOGFILE "$c -END OF FILE- $ec$NEWLINE";
     };
     print DOGFILE $eft; # Print EVEN if noTail=‘yes’
     close DOGFILE;
     print FILELOG " FILE CLOSED== ";
}
```

10 Trivial amendments

10.1 Print a section header

We now have the ability to label sections with pre-defined text (commented out). Remember that `$LINECOUNT` is an ugly global. We submit the section count `$SECTION`, and return this value incremented by one.

```
sub PrintSectionHeader
{ my($c, $SECTION, $ec, $nosections, $SECTIONTITLE);
     # $nosections 2011-12-18
    ($c, $SECTION, $ec, $nosections, $SECTIONTITLE) = @_;
  if (! $nosections)
  { if (length $SECTIONTITLE > 0)
       { $_ = $SECTIONTITLE;
         if (/\$\[SECTION\]/) # if contains section count
            { s/\$\[SECTION\]/$SECTION/;
            };
            print DOGFILE "$NEWLINE$c$_$ec$NEWLINE";
       } else
       { print DOGFILE "$NEWLINE$c - <Section $SECTION>"
                              . " - $ec$NEWLINE";
  };   };
  &PrintLogLine ("writing section [$SECTION]");
  $SECTION ++;
  return $SECTION;
}
```

The above code also replaces the text $[SECTION] (within the section header) with the actual section count.

10.2 Print LOG line

We often print to the log with a line number, so let's formalise this:

```
sub PrintLogLine
{ my($t);
   ($t) = @_;
   print FILELOG "\n  line $LINECOUNT: $t";
}
```

It makes code slightly more concise, at little cost.

11 Binary encoding and decoding

Responding to the need to write binary code from our TEX source:

11.1 UUdecode

We read the global file handle FIDO, mandating that the initial line is the 'begin verbatim' line. The line immediately after this must contain the UUencoded header line. The subsidiary routine UUdecodeLine returns not only decoded text, but also an error code. If the error code is less than zero, an error has occurred; if the error code is zero, then the subsequent line *must* be an end statement signalling the end of the UUencoded section!

```
sub Uudecode
{ my ($filename, $mode, @rslt);
  my ($line, $decoded, $err);
  $filename = "";
  $line = <FIDO>; # this should be \begin{verbatim} line:
  $LINECOUNT ++;
  if ($line !~ /\\begin\{verbatim\}/ )
     { &Alert ("Uudecode: no verbatim <$line>");
       return ("", "", "");
     };

  $line = <FIDO>; # MUST be header!
  $LINECOUNT ++;
  $line = &ChompLine($line);
  $line =~ /begin\s+(\d{3})\s+(.+)/;
  if (! defined $1)
     { # here write error!
       &Alert ("Uudecode: bad first UU line <$line>");
       return ("", "", "");
     };
  $mode = $1;
  $filename=$2;
  $err = 1; # -ve will signal failure

  while ( $line = <FIDO> )
    { # hmm what if extra 0xD ?
      last if (! defined $line); # ??
```

```
      $LINECOUNT ++;
      $line = &ChompLine($line);
      last if ($line =~ /^end/);
      if (! $err) # bad if err zero
         { &Alert ("Uudecode: end stmt not seen!<$line>");
           last;
         };
      ($decoded, $err) = UudecodeLine($line);
      # nb if $err is zero, next line must be /^end/!
      if ($err < 0)
         { # here could write error!
           if ($err == -1)
              { $err = "Bad line";
              }
           elsif ($err == -2)
              { $err = "silly length($decoded)";
              }
           elsif ($err == -3)
              { $err = "lengths don't match($decoded)";
              };

           &Alert ("Uudecode: error $err in <$line>");
           last; # terminate
         };
      push @rslt, $decoded;
    };
  return ($filename, $mode, join("",@rslt));
}
```

11.2 UUdecode line

Here's the routine that actually does the business of UUdecoding. We've kept this very simple, based on publicly available code.[7] On most UNIX/Linux systems, UUencoding and decoding should be readily available, but for DOS uuencoding, try e.g. this program.

```
sub UudecodeLine
{ my ($line) = @_;
  my ($charlen);
```

[7] http://search.cpan.org/src/ANDK/Convert-UU-0.52/lib/Convert/UU.pm

```
  my ($decoded, $ld);

  $line =~ /(.).*\‘*$/; # remove terminal backticks too!
  if (! defined $1)
     { return ("", -1); # dud line!
     };
   $charlen = (ord($1) - 32) & 077;
  if ($charlen == 0)
     { # ie terminal line with single backtick:
       # no error, but END!
       return ("", 0);
     };
  if (($charlen > 45) || ($charlen <0))
     { return ("$charlen($1)", -2); # bad length
     };

 # convert to number, then count of characters encoded;
  $decoded = unpack("u", $line); #uudecode!
  $ld = length $decoded;
  if ($ld != $charlen)
     { return ("$ld:$charlen:$decoded", -3); # length doesn’t match!
     };
  return ($decoded, 1); # success!
}
```

See the archaic use of octal. But it works.

A brief note on uuencoding/decoding

This description assumes you understand hexadecimal and ASCII. A uuencoded file consists of:

1. A header line;
2. A body;
3. A trailer line.

All other lines must be ignored Lines may end with 0x0D, 0x0A, or any combination of the two (ie carriage return and/or line feed). From now on we’ll call the end of line character(s) the ’endline’. Conventionally this should be *encoded* as simply 0x0A. Here are the details:

1. The header line. This contains three items *separated* by spaces (0x20):

   (a) The five character string 'begin' (no quotes around it)

   (b) Three digits, each in the range 0..7 i.e. an octal number

   (c) The file name in ASCII (potential for trouble here!)

   The header line terminates with an endline.

2. The body. This contains one or more lines, each ending with an endline. For all but the last data line, there should be 62 characters in a line:

   (a) A single character, usually the ASCII character M

   (b) 60 characters representing an encoded string

   (c) The endline

   For the last line, some variation is permitted: The first character can be in the range 0x20 to 0x5F. There's a FURTHER CHECK: the very last line of the body does NOT contain data and is simply made up of a single backtick character.

   In all cases:

   (a) The first character represents the number of encoded characters, with 0x20 added! This is why all lines but the last should start with M: they contain 45 encoded characters (hex 0x3D). The ASCII representation of M is 5D, ie 0x3D + 0x20.

   (b) Encoding of characters is done three-at-a-time. If there are less than three characters, we pad with hex zero (0x0). Encoding is as follows:

      i. Divide the 3 bytes (3*8 = 24 bits) into four groups of 6 bits, working from left to right;

      ii. This gives us four numbers between 0x0 and 0x3F;

      iii. To each number, add 0x20, giving numbers in the range of 0x20–0x5F;

      iv. Write the numbers as four ASCII characters to the output file.

      In other words, we output characters in the set of:

   `!"#$%&'()*,-./:;<=>?@[\]^_` as well as plus, space, 0–9 and A–Z.

3. The trailer line. This starts with the three character string 'end' (No quotes).

There are some frills:

- Also permissible are ASCII characters $>$ 95 (5Fh) but only the rightmost 6 bits are relevant.
- The three digit number on the first line is the file mode (read / write / execute) The first number is the octal representation of the read permission of the file, the next the write permission, and finally the execute permission.
- Because some mailers used to strip off terminal blanks, it is usual (and perhaps wise) to pad such lines (with at least one terminal blank) with supplementary junk characters. The usual character is the backtick, 0x60, which has the added advantage that it translates to the otherwise illegal value 0x0 when the high bits are masked off.

12 Change log

12.1 Changes in version 2.0

The major changes in version 2.0 were related to the ability to shift sections down below other sections (defer writing of certain sections), waiting for all dependencies to be fulfilled before writing the code. Names are only resolved when all of the dependencies of that section have been met.

It would be possible to keep a record of 'names already resolved' but this is a little silly. We are only interested in deferring the writing of code, so there's little point in bookkeeping to this extent. Just leave out the dependency if it's already resolved!

If a child has no name, then the corresponding PENDINGNAME element is `''`. We could decrease memory requirements for CHILDREN by deleting array elements once written; we might benefit from even removing all corresponding elements entirely (as we will resolve the name immediately and the element in DEPENDENCIES is of course empty as well).

We also introduced the concepts of line concatenation, chomping off line feeds and subsequent leading spaces.[8]

Another amendment was allowing alteration of the initial comment string (formerly newComment, now startComment), the noWarn option to suppress irritating warnings (especially with multiple file writes), and OPTIONAL statements for a debugging version.

We also introduced insertion of binary (uuencoded) files, by allowing the user to say `newTarget='uudecode'`. Usage of the Uudecode function is:

```
($filename, $mode, $outstring) = Uudecode();
```

12.2 Changes in version 2.1

1. We allow file start and end code. This is really for PHP, where such code is vital, but also for included HTML. The directive we use, for example startFile='<?php' must be in the same line as the file name specification, but we can specify the endFile='?>' directive any time before we terminate the file for which we require such terminal code. The length of either can be specified as zero using startFile='' or endFile=''. **Note** that after each file is written, these text strings are reset to the null string, so the startFile and endFile values must be specified for *each file* in which they are used!

[8] When using the WinEdt text editor, the default is to trim trailing spaces, which can be rather irritating. You have to uncheck Options|Preferences|defaults|Trim spaces, or in already created documents uncheck Document|document settings|trim spaces.

2. We removed use of `=cut`, as we had problems with some Perl versions, notably v5.8.

3. We set the focus to the 'Wag' button, so on running the program with a file argument, things are ready to run.

4. We allowed comment closure (as in HTML, older versions of C), along the lines of *endComment*='–>' for HTML. The default closing comment is an empty string. Note that unlike startFile and endFile, startComment persist until changed. (As of version 4.0.0, applies to endComment too).

5. We fixed a problem with multiple dependencies, where a section was printed after just one dependency was resolved owing to an error in testing the remaining list of dependencies!

6. In minor version 2.1.1, if the user declines to overwrite a given file, we don't abort the whole Dogwagger process — instead, we write the data to the file 'JUNK.JUNK' and carry on! Note that we overwrite this file, and don't append, so if you decline to overwrite multiple files, only the last will be kept.

7. In minor version 2.1.2, we remove the text `--` from the end file and section annotations, to prevent conflict with HTML formatting checks.

12.3 Changes in version 3.0

Okay, basically just got rid of Tk, and rewrote things to accommodate this.

Version 3.0.1

Fixed error with line counts. Note that in LyX, we can really mess things up by having a line continue with the ERT containing a verbatim statement, as this is now missed by DogWagger.

Also print the name of the file being processed to stdout, and put in a carriage return before asking about "Overwrite...?"

12.4 Version 4.0

A few 'issues', sorted out on 2011-12-18 – 2011-12-19. This started as a minor rewrite and expanded a bit, so the new version number seems justified.

1. Because of issues with chomp, the section that used this to concatenate multiple lines into one (`oneLine='yes'`) has been rewritten.

2. Allied to the preceding issue, different source documents will use different conventions for a new line. Conventions are:

   - Macintosh (obsolete) 0D (`\r`)
   - Unix/Linux, and Mac OS X: 0A (`\n`)
   - MS Windows 0D,0A (`\r\n`)

   We will generally need to preserve the characteristics of a document faithfully, but should provide the option of recoding using a different convention (`newLine='whatever'`).

3. If we had `noWarn='yes'` before any other parameterised statement, this failed, because we used greedy matching. Fixed.

4. We really do need some mechanism of suppressing the header comments that are inserted by DogWagger. I thus introduced the option:
   `noHeader='yes'`

5. While we're about it, let's add `noSections` and `noTail` too.

6. Because suppression of header, sections and tail is against our basic philosophy, and we don't want any 'surprises', we will reset these options to the default on closing any written file. The problem is that up till now, we have read several file-related parameters "on the fly". These include:

   - startComment / newComment
   - include
   - noWarn
   - endComment
   - startFile
   - endFile
   - noHeader
   - noTail

   We now do the honourable thing, and read these parameters (ReadTargetParams) when we encounter newTarget! This does raise the possibility that someone might wish to change these parameters within a file. Hmm.

7. It's nice to have `perl Dogwagger405.pl --help` produce a list of options. (done).

8. We now warn about unrecognised options. This includes default options that are automatically reset and thus should not be specified.

9. Fixed the bug that permits commented-out verbatim statements from being included.

10. Having newComment is quite confusing, so I deprecate this, and recommend simply using startComment consistently.

11. The startComment / endComment retention predisposes to an unwanted endComment being retained after startComment has changed. We should thus automatically turn the endComment to '' if the startComment is altered BUT the endComment is NOT! This change has been made in ReadTargetParams.

12. Line counts now work after deferred code and uudecoding.

13. We need the option to write to a different log file (distinct from the usual 'WAGLOG.LOG'). I have therefore introduced the command line option of log=logfilename.

I've also prettied up the code to get rid of the nasty text overruns in the PDF document, and made some minor amendments to neaten up the text.

Version 4.0.1

1. Change from LaTeX to LyX, with a few frills (and a logo);

2. A major revision of the documentation;

3. Less picky about case in searching for "dogwagger";

4. Cleaned up checking for 'uudecode' string;

5. Introduced ability to arbitrarily turn section headings on and off (not only at file start);

6. Maintain (and print) counts of number of sections used, files encoded, etc.

7. Prettied up (and standardised) log output.

Version 4.0.2

A few bugs and amendments.

1. noSections = 'yes' fails in 4.0.1. This is because when I rapidly added the code to permit suppression/activation of individual sections, I failed to anticipate the following problem: the command is excised; section suppression is turned on but then immediately off (by default) at the start of the file; and the command is no longer visible when the new file parameters are read. Fix by checking for newTarget in Section 5.7.

2. Section 2.7 rewritten to accommodate a message about the best location of the very first Dogwagger line (in the LaTeX preamble).

3. Some reformatting. The use of Sans-serif throughout is just too in-your-face, so I've changed to Roman. The base font size is a bit problematic, so some of the sections have been rendered as \footnotesize to make them fit.

4. [Note that as things stand, uuencoded files cannot be stored to subdirectories. Should we fix this?]

Version 4.0.3

Oops. We really need to rethink the poor code in Section 5.7. The silliness of this code reflects the harm that initial bad coding inflicts on subsequent versions.

1. Some reformatting of the offending section;

2. Changed test for "unrecognised commands" to allow for the possibility of (illegal) whitespace in between **command** and **=** and **'parameters'**.

3. Search the first 100 lines for the initial Dogwagger line (LyX can put in a lot of stuff).

Version 4.0.4

On 15 May 2012, Mark Ellison insightfully pointed out that some authors may wish to intersperse file components, specifying parts of *different files* 'out-of-order'.[9] Even better, he made small amendments that address the problem. These are all labelled 'v4.0.4 (tme)' in the preceding code:

1. Re-entrancy is set up in Section 3.2 using an associative array;

[9]The program actually does allow you to defer a section until after the current file closes, but this is clumsy and generates an error message.

2. The file count is only bumped if we are opening a new file (Section 5.7.1);

3. When we open the target file (Section 9.1) we *append* if we've previously opened this file during the current run.

There are a few caveats. The obvious one is that you will generally need to specify all of the necessary header parameters every time you open the file, for example the comment format (unless the files you are flicking between use the same format). A second and slightly more ominous 'gotcha' concerns `startFile` and `endFile`. The startup code (for e.g. PHP or HTML) should only be specified the first time you open the file, and the end code should only be specified the last time you open the file to append code.

Version 4.0.5

The sole change: Dogwagger now returns error code 99 on failure—See page 15.

A Appendix: GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you". Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program. You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.) These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.) The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable. If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either

of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND / OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND / OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

(END OF TERMS AND CONDITIONS)

B Appendix: UUencoding

In older versions I included a tiny *uuencoded* Windows uuencoding program here—Dogwagger would automatically extract it. The .com program no longer works in modern versions of Windows. Instead, to uuencode `FOO.bar` in a single console line say:

```
certutil -encode FOO.bar tmp.b64 && findstr /v /c:- tmp.b64 > FOO.b64
```

This trick comes from Igor Kromin; it uses `findstr` to remove the "certificate" lines at the start and end of the temporary file. To remove the temporary file `tmp.b64` add `&& del tmp.b64` to the end of the above line.

To uuencode in Linux try:

```
base64 FOO.bar > FOO.b64
```