Syntax Highlighting Modular Build System for NoWeb (Users' Guide)

Thomas J. Moore

Version 3.13.183
November 26, 2012

Building packages based on `build.nw`

Building from scratch has a number of system dependencies:

- GNU (`http://www.gnu.org`) make, or a clone. In particular, Daniel Pfeiffer's makepp (`http://makepp.sourceforge.net`) works as well.
- Some standard UNIX tools: GNU sed, POSIX awk, grep, cat, and others. The only non-standard tool is tac, which can be replaced (see makefile.config).
- Perl. Any version will probably do.
- Norman Ramsey's NoWeb (`http://www.eecs.harvard.edu/~nr/noweb/`). If you wish to develop code using this system, you will need to understand NoWeb as well (it's not difficult). If you don't want to generate documentation, you can replace the noroots command using makefile.config; a replacement for notangle is given below.
- For PDF output, you will need good, working LaTeX installation. There are a number of bundles which include almost everything you'll ever need. In particular, the framed package must be up-to-date. Note that XeTeX is supported as well. For PDF syntax highlighting, the listings package may be used.
- For HTML output, you can either use the simple LaTeX-to-HTML converter included with NoWeb, or you can use tex4ht (`http://www.cse.ohio-state.edu/~gurari/TeX4ht/`).
- For syntax highlighting in HTML using NoWeb's LaTeX-to-HTML converter, or if you are dissatisfied with the listings package, you will need a separate highlighter command. Currently, GNU source-highlight (`http://www.gnu.org/software/src-highlite/`) and André Simon's highlight program (`http://www.andre-simon.de`) are supported. For the former, only recent versions have been tested. For the latter, version 2 or version 3 is supported.

You may have been provided with a source tarball, in which case that should eliminate the need for NoWeb, Perl, awk, GNU sed, and others. You simply need to view and edit `makefile.config`, and then type `make` or `make install`.

To build, place the NoWeb source files into their own directory (optionally extracting them from the document you are reading first), extract the makefile, and make using GNU make:

```
# if your PDF viewer supports attachments, save the attachment
# otherwise, use pdfdetach:
pdfdetach -saveall build-doc.pdf
# or pdftk:
pdftk build-doc.pdf unpack_files output .
# then extract the makefile and build
notangle -t8 build.nw > makefile
```

```
make install
```

If NoWeb is not available, but perl is, the following code can be copied and pasted into a text file using a reasonable document viewer and made executable as an imperfect, but adequate `notangle` replacement:

```
#!/bin/sh
#!perl
# This code barely resembles noweb's notangle enough to work for simple builds
eval 'exec perl -x -f "$0" "$@"'
  if 0;
my $chunk = '*';  my $tab = 0;

while($#ARGV > 0 && $ARGV[0] =~ /^-.*/) {
  if($ARGV[0] =~ /^-R/) {
    $chunk = $ARGV[0];
    $chunk =~ s/-R//;
  } elsif($ARGV[0] =~ /^-t(.*)/) {
    $tab = $1;
  }
  shift @ARGV;
}
my ($inchunk, %chunks);
while(<>) {
  if(/^<<(.*)>>=$/) {
    $inchunk = $1;
  } elsif(/^@( |$)/) {
    $inchunk = "";
  } elsif($inchunk ne "") {
    $chunks{$inchunk} .= $_;
  }
}
my $chunkout = $chunks{$chunk};
while($chunkout =~ /(?:^|\n)(|[^\n]*[^\n@])(?:<<((?:[^>\n]|>[^>\n])*)>>)/) {
  my $cv = $chunks{$2};
  my $ws = $1;
  $cv =~ s/\n$//;
  $ws =~ s/\S/ /g;
  $cv =~ s/\n/$&$ws/g;
  $chunkout =~ s/(^|[^@])(<<((([^>\n]|>[^>\n])*)>>)/$1$cv/;
}
$chunkout =~ s/@(<<|>>)/\1/g;
$chunkout =~ s/((^|\n)\t*) {$tab}/$1\t/g if($tab gt 0);
print $chunkout;
```

Additional build configuration can be done in `makefile.config` before installing (in particular, the install locations). This file can be generated using either `make makefile.config` or `make -n`. On the other hand, to avoid having to modify this file after cleaning, `makefile.config.local` can be created for this purpose instead.

The following are the build parameters for `build.nw` itself:

```
# Installation prefix to apply to all install locations
DESTDIR:=
# Installation directory for binaries
BIN_DIR:=/usr/local/bin
# The name of the file containing the makefile
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#BUILD_NOWEB:=build.nw
# The name of the source files
# Any time you change this, makefile.* should be removed and rebuilt
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#NOWEB:=$(wildcard *.nw)
```

```
#Set to override the automatically determined project name
#PROJECT_NAME=
# Set to -L for #line, but lose code indentation
USE_LINE:=-L
# Where to find the non-standard tac command (GNU coreutils)
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#TACCMD:=sed -n -e '1!G;h;$$p'
# The noroots command is used to extract file names from $(NOWEB)
# The following works well enough in most cases if noweb is missing
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#NOROOTS=sh -c "sed -n '/^<<.*>>=\$$/{s/=\$$//;p;}' \$$* \
#                                                  | sort -u" /dev/null
NOROOTS=noroots
# The PDF command to use: must be pdflatex or xelatex.
LATEXCMD=pdflatex
# note: to use attachments with xetex, the following may be necessary:
#LATEXCMD=env DVIPDFMXINPUTS=.:\$$TEXMF/dvipdfmx/ xelatex
# if non-blank, specify which highlighter to use:
#    highlight-2    named highlight, acts like highlight-2.x
#    highlight-3    named highlight, acts like highlight-3.x
#    source-highlight named source-highlight (probably 3.x)
# if blank, autodetect
HLPROG_TYPE=
# Highlight theme for PDF/HTML formatted source code
# Blank uses default highlight style; prefix w/ list: to use listings package
HL_THEME:=
# Program to use for generating HTML formatted source code
# Set to tex4ht or l2h
HTML_CONV:=l2h
```

Creating packages based on `build.nw`

Users of this document fall into at least two categories: ones who wish to create documents that use this build system, and ones who wish to build packages which were created using this build system. This section is for the former. Instructions for the latter were included on the first page, but should also be included in any document using this system:

```
\input{build-doc.tex} %%% doc
```

Of course the instructions for document users are required to build and test the project for document creators as well. After creating the makefile, you can use `make` to build and test. The standard targets are `all` (the default), `bin`, `install`, `doc`, `misc`, `clean`, `distclean`, `count`, `check`, `test-bin`, and `test`. In particular, the `check` target can help ensure that changes to a source file will not result in weird errors due to lost chunks or mistyped chunk names.

The first few lines of a NoWeb file are LaTeX code. This must at least include the `\documentclass` directive. It can also include the build system's preamble in the place marked by `%%% latex preamble`. Copying the beginning of this document to start with would not be a bad idea:

```
% Build with noweb:
```

```
%  notangle -t8 build.nw > makefile
%  make
\documentclass[twoside,english]{article}
\usepackage[letterpaper,rmargin=1.5in,bmargin=1in]{geometry}
%%% latex preamble
\RCS $Id$
\RCS $Revision$

\begin{document}

\title{...}
\author{...}
\date{Revision \RCSRevision}

\maketitle
```

Dependencies on other files should be listed somewhere (usually near the top), as well. They are specified by comments of the form `%%% requires` *name* on lines by themselves. Only direct dependencies need to be listed, as a full tree will be generated. The `.nw` extension is optional. For example, somewhere in your project, you will need to depend on this file:

```
%%% requires build
```

Chunks defined in a document extend chunks defined in documents on which it depends. Chunks defined in other documents can be used, as well. In both cases, they will not be weaved correctly. Chunks which are extended will be indicated with the original source file in parentheses at the start of the chunk name; this format is used mainly to move external referneces to the top of the chunk index and sort them by source name. However, the first extension will have no "previous chunk" link, rather than pointing to the source document's definition. Uses are also indicated with the original file name, but they always claim to be undefined. This is somewhat alleviated by the fact that undefined references are labeled with the word imported rather than undefined, but the only way to remove that indicator is to define the chunk elsewhere for extension (e.g. in an appendix) as neutral as possible (blank or a comment in the source language). This can be disabled for user-level documentation by adding the commment:

```
%%% no-ext-ref
```

Within the document, several extensions can be used:

- Language definitions: the listing package's `\lstset{language=`*lang*`}` directive is used to set the language for any subsequent code chunks. The default language is C.
- Hidden sections: any figures or other files which are normally included in the document in a processed form can be stored in-line in the NoWeb document in their raw form in hidden sections. Each hidden section is delimited by `<!-->` on its own line at the start, and `<-->` on its own line at the end. This is intended to be used as follows:

```
% hidden section - notangle chunks that shouldn't appear in printed output
% mainly for diagrams, which are extracted and included in visual form.
\begin{rawhtml}
<!-->
\end{rawhtml}
\iffalse

... hidden material ...
```

```
\fi
\begin{rawhtml}
<-->
\end{rawhtml}
```

These are also provided as reinsertable documentation chunks, so the following is equivalent:

```
\input{begin-hidden.tex} %%% doc
```

... hidden material ...

```
\input{end-hidden.tex} %%% doc
```

NoWeb doesn't actually strip these out — iffalse does that. Because NoWeb generates cross-reference information and the chunk index after the last chunk, the last chunk may not be commented out in this way. If you see a message like "LaTeX Warning: The are no \nowebchunks on input line ...", there are eiher no code chunks at all, or the last code chunk was commented out.

- Syntax highlighted, fully extracted code chunks: a code chunk can be repeated in e.g. a users' guide appendix, fully extracted and syntax highlighted, by using a special `\input` directive: `\input{`*chunk-name*`.tex} % `*language*. This directive will be replaced by the contents of the named chunk. For example, the configurable variables are included here using:

```
\input{makefile.config.tex} % make
```

```
# Installation prefix to apply to all install locations
DESTDIR:=
# Installation directory for binaries
BIN_DIR:=/usr/local/bin
# The name of the file containing the makefile
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#BUILD_NOWEB:=build.nw
# The name of the source files
# Any time you change this, makefile.* should be removed and rebuilt
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#NOWEB:=$(wildcard *.nw)
#Set to override the automatically determined project name
#PROJECT_NAME=
# Set to -L for #line, but lose code indentation
USE_LINE:=-L
# Where to find the non-standard tac command (GNU coreutils)
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#TACCMD:=sed -n -e '1!G;h;$$p'
# The noroots command is used to extract file names from $(NOWEB)
# The following works well enough in most cases if noweb is missing
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#NOROOTS=sh -c "sed -n '/^<<.*>>=\$$/{s/=\$$//;p;}' \$$* \
#                                                  | sort -u" /dev/null
NOROOTS=noroots
# The PDF command to use: must be pdflatex or xelatex.
LATEXCMD=pdflatex
# note: to use attachments with xetex, the following may be necessary:
#LATEXCMD=env DVIPDFMXINPUTS=.:\$$TEXMF/dvipdfmx/ xelatex
# if non-blank, specify which highlighter to use:
#    highlight-2    named highlight, acts like highlight-2.x
```

```
#    highlight-3    named highlight, acts like highlight-3.x
#    source-highlight named source-highlight (probably 3.x)
# if blank, autodetect
HLPROG_TYPE=
# Highlight theme for PDF/HTML formatted source code
# Blank uses default highlight style; prefix w/ list: to use listings package
HL_THEME:=
# Program to use for generating HTML formatted source code
# Set to tex4ht or l2h
HTML_CONV:=l2h
```

One use of this feature which should be in every document is to list the RCS/CVS/subversion ID tag of all source files from which the document was built:

> This document was generated from the following sources, all of which are attached:
> `\input{Sources.tex} % txt`

- Syntax highlighted C prototypes: the prototype of a C function can be repeated as a highlighted code chunk using a special comment: `%` *function-name* `prototype` prints a reformatted version of the prototype extracted from `cproto.h`, which is automatically generated for all C files. Any prototypes which would not be placed there can be manually added using the ⟨*C Prototypes*⟩ chunk; all prototypes from the chunk must consist of exactly one line of text.

- Repeated documentation: a separate NoWeb-like macro facility which only does substitutions can be used to repeat documentation in e.g. a users' guide in an appendix. Such documentation chunks are delimited by `% Begin-doc` *chunk-name* and `% End-doc` *chunk-name* at the beginning of a line. They are inserted in place of any occurence of `\input {`*chunk-name*`.tex} %%% doc` at the beginning of a line.

- Simple input directives: a file can be included verbatim using `\input{`*file*`.nw}`. This actually substitutes the file in place rather than doing an actual include so that weaving and tangling operate correctly.

- Documentation attachments: any files which should be attached to the PDF or HTML output can be attached using the `embedfile` package's `\embedfile` directive.

Somewhere in the document, the code chunk index should be added. Right now, an identifier index is not supported.

```
\section{Code Index}
\nowebchunks

% note: no identifier indexing is done right now

\begin{rawhtml}
<!-->
\end{rawhtml}
%\vspace{1ex}
%\hrule
%\vspace{1ex}
\begin{rawhtml}
<-->
\end{rawhtml}

%\nowebindex
```

Of course the document must be ended with `\end{document}`, like any LaTeX document.

```
\end{document}
```

Once the documentation has been built, the PDF output needs to be checked for overflows. Horizontal overflows in code chunks are hard to detect, as they do not always generate `Overfull \hbox` messages. They can be corrected by reformatting the source code. Vertical overflows as the result of problems with the `framed` package can be detected by `Overfull \vbox` messages, and can be corrected by prefixing chunks which are not broken at the end of a page with a `\break` directive. Naturally, any edits to the document will require rechecking all `\break` directives to see if they are still necessary, and possibly to add more.

Since I write a lot of C, a few special tricks are available for C programs. In addition to the C-specific chunks described below, a special header is generated, `cproto.h`, which contains all exported prototypes, except for prototypes already explicitly included in headers. This is always included after ⟨*Common C Includes*⟩ in the ⟨*Common C Header*⟩. In order to facilitate writing static functions out of order, a static prototype file is generated for each C file as well. This is included right after `cproto.h`, unless the comment `// static_proto` is found on a line by itself, in which case it is placed after that comment. That way, the static prototypes can be placed after any private data type declarations. As mentioned above, `cproto.h` is used for API documentation as well.

Several internal variables are meant to be set or expanded in extensions to ⟨*makefile.vars*⟩:

- `C_POSTPROCESS` is a pipeline to apply to all automatically extracted C files. This is a function taking the file name as an argument (which can be ignored if there are no file-specific transformations). The return value (or value of the plain variable, since there is no difference between functions which ignore their arguments and variables) should be a pipeline element, starting with the pipe symbol. For example:

  ⟨*makefile.vars*⟩=
  ```
  C_POSTPROCESS+=|sed '1i\copyright blurb'
  @
  ```

- The `EXTRA_CFLAGS` variable should be extended with required options; that way, the user can replace `CFLAGS` with just optimization and debugging flags.

- The `EXTRA_LDFLAGS` variable should be extended with required options; that way, the user can replace `LDFLAGS` with just optimization and debugging flags.

Finally, extensions to the build system can be made by adding code chunks; they will be appended to existing values in the order of the tree, from the most depended on file to the least depended on file (i.e., anything depending on `build.nw` will append values *after* the values set in `build.nw`):

Basic build rules:

- ⟨*makefile.config*⟩: add user-configurable variables to the makefile.
- ⟨*makefile.vars*⟩: add non-configurable variables to the makefile.
- ⟨*makefile.rules*⟩: add rules to the makefile.
- ⟨*Install other files*⟩: add installation commands.
- ⟨*Clean temporary files*⟩: add commands to clean temporary files.
- ⟨*Clean built files*⟩: add commands to clean built files.
- ⟨*Plain Files*⟩: add plain files to build; always add a backslash to the end of each added line. If additional processing other than simple extraction is required for any file, it should instead be added to ⟨*makefile.rules*⟩ as a separate rule and a dependency for the `misc` target. If any plain files are not in the NoWeb source, but instead are distributed separately, they should instead be added to the `ATTACH_EXTRA` variable, to be attached to the documentation. That variable can also be a make function taking the name of the file being attached to as an argument.

- ⟨*Plain Build Files*⟩: add plain text files to be built when required as a dependency; always add a backslash to the end of each added line. This is not added to any explicit target. It is merely a shorthand to create a rule for extracting it from NoWeb and cleaning it up afterwards. It is intended for build support files that are not meant to be distributed.
- ⟨*Script Executables*⟩: add plain text files to build as executables; always add a backslash to the end of each added line. Like ⟨*Plain Files*⟩, no processing is done by the default rules.
- ⟨*Build Script Executables*⟩: add plain text files to build as executables when required; always add a backslash to the end of each added line. This is not added to any explicit target. It is merely a shorthand to create a rule for extracting it from NoWeb and cleaning it up afterwards. It is intended for build support files that are not meant to be distributed.
- ⟨*Test Scripts*⟩: add plain text files to build as executables, but only for the `test` and `test-bin` targets. The `test` target will execute these scripts without any interpretation other than return code checks.
- ⟨*Test Support Scripts*⟩: names of executable scripts to build for the `test-bin` target. These scripts are not run, but are guaranteed to be built before running any tests.
- ⟨*Additional Tests*⟩: tests that are more complicated to run than simple scripts. These are makefile actions.
- ⟨*Build Source*⟩: add files to the NoWeb-free tar distribution; always add a backslash to the end of each added line. In particular, any specially processed plain files with their own rules should be added to this list.
- ⟨*Files to Count*⟩: files to count as part of the `count` target; always add a backslash to the end of each added line. Anything added to the ⟨*Build Source*⟩ should probably be added here, as well.
- ⟨*Executables*⟩: additional specially-built execuatables, other than scripts and C executables.

C-Specific rules

- ⟨*C Files*⟩, ⟨*C Headers*⟩: C files which are NoWeb roots are automatically found, but generated files are not. Add these with these chunks, terminated by a backslash.
- ⟨*C Executables*⟩: C files are either the name of an executable with the `.c` extension, or the name of a library member, with `.o` replaced by `.c`. The executables are listed here, without the `.c` extension, but with a trailing backslash on every line. Technically, this could be considered a list of generated executables instead, since there is no rule requiring that these be C.
- ⟨*C Test Executables*⟩: names of executables to build for the `test` and `test-bin` targets. See ⟨*C Executables*⟩ and ⟨*Test Scripts*⟩ for details.
- ⟨*C Test Support Executables*⟩: names of executables to build for the `test-bin` target, as explained by ⟨*C Executables*⟩ and ⟨*Test Support Scripts*⟩ for details.
- ⟨*C Build Executables*⟩: names of executables to build when required by other dependencies. See ⟨*C Executables*⟩ and ⟨*Build Script Executables*⟩ for details.
- ⟨*Library* `name` *Members*⟩: A library named `name` will be built using objects listed in these chunks. `name` can of course be anything appropriate. Unlike the chunks which expand into specific `make` variables, no backslashes are required or allowed at the end of each line.
- ⟨*For each reinserted C prototype*⟩: add a transformation for the modified function name (`$mf`). Typically, this will involve invoking the C preprocessor with a few standard include files, followed by the function name. For example:

⟨*makefile.vars*⟩=
```
export MY_CPP=$(CC) $(CFLAGS) $(EXTRA_CFLAGS) -E
@
```

⟨*For each reinserted C prototype*⟩=
```
    mf="`printf '⟨Common C Includes⟩\\n%s' \"\$mf\" |
            eval \"\$MY_CPP -\" | tail -n 1`"
@
```

Boilerplate

- ⟨*Common NoWeb Warning*⟩: a boilerplate comment for the top of script files; extend with `# $Id$`.
- ⟨*Version Strings*⟩: A C string containing the version ID of all contributing source files; extend with `"$Id$\n"`.
- ⟨*Sources*⟩: A simple text chunk containing the version ID of all contributing source files; extend with just `$Id$`.
- ⟨*Common C Warning*⟩: the same comment enclosed in a C comment; not meant to be extended
- ⟨*Common C Includes*⟩: a list of `#include` directives for files safe to be included by all C source
- ⟨*Common C Header*⟩: stuff safe for the top of every C file, including the warning, common includes, static version string, and automatically generated prototypes. Prototypes for static functions for the file are included as well, unless a comment of the form `// static_proto` is found on a line by itself; in that case, the static prototypes will be included there instead.
- ⟨*Common Perl Prefix*⟩: stuff to insert at the top of perl executables. This includes the NoWeb warning, a portable hash-bang prefix, and `use strict`.

Highlighting Assistance

- ⟨*Known Data Types*⟩: comma-separated list of data types to emphasize; end each line (including the last) with a comma and a percent-sign.
- ⟨*Translate listings source type to source-highlight source type*⟩, ⟨*Translate listings source type to highlight-3 source type*⟩: add new cases to a shell case switch to convert listings language names to highlighter language names.
- ⟨*Mangle aux files*⟩ can be extended to further process the output from the previous LaTeX run. It is makefile action text for a single command, so semicolons and backslashes need to be after every line, and dollar signs need to be doubled. The make variable `$*` is the base name of the file being processed. Setting the shell variable `$rerun` to non-empty will force LaTeX to be rerun without checking the log file. These actions are run after the index and bibliography have been generated. To run something before that, extend ⟨*Mangle aux files before bbl and ind*⟩ instead.

See the beginning of this document for prerequisites; the same applies to development as well. Better yet, read the original source for comments on what is needed, and why.