1 Introduction

The `dsbweb` system is a code documentation system inspired by Knuth's `WEB` system and its offshoots, as well as by systems like Javadoc and Doxygen. `dsbweb` extracts markup (LaTeX or Markdown) embedded into structured comments in C/C++, FORTRAN, MATLAB, Lua, or shell codes. The code and documentation therefore live together in a single file, such as this one, which can be compiled into an executable without special processing. To generate documentation, one runs `dsbweb` on the same file; the tool then spits out a documentation file which contains ordinary markup text extracted from comments, and code embedded in a specified environment. The processor knows whether it is looking at documentation markup or at code by the presence of the command sequences `@t` or `@T` (TeX) and `@c` (code) in the structured comments. By default, the processor is in quiet mode (`@q`); in this mode, nothing at all is sent to the documentation file.

We already have Doxygen, Javadoc, noweb, nuweb, and others; what's the use of another documentation tool? There are a few features I want in a documentation tool, and none of the tools that I could find had them all. These features are:

1. I want to selectively *turn the thing off*. When I read my own documentation, I do not care to see every include file that I've used, nor every hack and interface adapter needed to get one piece of code to talk to another. I prefer to be able to concentrate on the pieces that are in some way interesting, unusual, or (alas) tricky. If I want all the messy details, I'll look at the raw code.

2. I want documentation in comments. One of the major drawbacks of Knuth's `WEB` and its successors is that they introduce a new type of source file from which regular source files are derived. My favorite source processing tools – Emacs modes, `indent`, etc. – don't always work with those web file types. I prefer the Javadoc / Doxygen model for this reason.

3. I write numerical software, and I want to be able to put mathematics into my documentation. I have a bias toward LaTeX.

4. I want to document C/C++, FORTRAN 77, and MATLAB. I use a mixture of languages, and I like to be able to document more than just languages in the C family. I also want to be able to add support for new languages without too much difficulty.

5. I want to be able to run the tool anywhere without much fuss. It shouldn't depend on external packages that I might have to install (or, more to the point, that I might have to tell someone else how to install).

This document describes the `dsbweb` processor code. It is also generated using the `dsbweb` processor using the commands:

```
    dsbweb dsbweb.c -d dsbweb.tex
    pdflatex dsbweb.tex
```

2 Help output

```
char* DSBWEB_HELP =
    "Syntax:\n"
    "  dsbweb [-help] [-md] [-p preamble] [-d output] [-o output.tex]\n"
    "    [-e envname] [-c] [-m] [-f] [-l] [-mb] [-list] files\n"
    "\n"
    "Flags:\n"
    "  -help  See this message\n"
    "\n"
    "  -md    Use Markdown mode rather than default (LaTeX)\n"
    "  -o     Send all the output to output rather than to defaults\n"
    "  -d     Send all the output to output.tex, along with a preamble\n"
    "         and begin/end document commands.\n"
    "  -p     Specify a TeX input file to be included in the preamble.\n"
    "         This is only useful in conjunction with the -d flag.\n"
    "  -e     Specify an environment (e.g. listings) for typesetting code.\n"
    "         The default is verbatim.\n"
    "\n"
    "  -c     Specify that following files are in C\n"
    "  -m     Specify that following files are in MATLAB\n"
    "  -l     Specify that following files are in Lua\n"
    "  -sh    Specify that following files are in shell\n"
    "  -lisp  Specify that following files are in LISP\n"
    "\n"
    "  -mb    Process the following files as MATLAB batch files\n"
    "  -list  Print the contents of MATLAB batch files\n"
    "\n";
```

3 Determining languages and base names

We determine which language we're using by looking at the file name. We identify Makefiles by the last eight characters of their name (as shell type), and everything else by extension: `*.m` is MATLAB, `*.(f|F)` is FORTRAN, `*.lua` is Lua, `*.sh`, everything else is C/C++.

```
#define DSB_NOLANG -1  /* Wildcard language type */
#define DSB_C   0      /* C/C++/Java language */
#define DSB_M   1      /* MATLAB language */
#define DSB_MB  2      /* MATLAB language with files batched */
#define DSB_F77 3      /* FORTRAN 77 */
```

```
#define DSB_LUA 4      /* Lua */
#define DSB_SH  5      /* Shell / Makefiles */
#define DSB_LISP 6     /* LISP dialects */
#define DSB_MB_LIST 7  /* Special command - list entries in a batched file */

int language_type(char* fname)
{
    char* p = fname + strlen(fname);
    for (; p != fname && *p != '.'; --p);
    if (strlen(fname) >= 8 &&
        strcmp(fname+strlen(fname)-8, "makefile") == 0 ||
        strcmp(fname+strlen(fname)-8, "Makefile") == 0)
        return DSB_SH;
    else if (p == fname)
        return DSB_C;
    else if (strcmp(p, ".m") == 0)
        return DSB_M;
    else if (strcmp(p, ".f") == 0 || strcmp(p, ".F") == 0)
        return DSB_F77;
    else if (strcmp(p, ".lua") == 0)
        return DSB_LUA;
    else if (strcmp(p, ".sh") == 0 || strcmp(p, ".csh") == 0)
        return DSB_SH;
    else if (strcmp(p, ".l") == 0 || strcmp(p, ".scm") == 0)
        return DSB_LISP;
    else
        return DSB_C;
}
```

Similarly, we determine the base name for a file by figuring out the extension and replacing it with the extension specified in `ext`. If there is no extension, we append `ext` to the entire filename. In the default LaTeX mode, we use `.tex` for the extension; in Markdown mode, we use the `.md` extension.

```
FILE* open_markup_output(char* fname, const char* ext)
{
    FILE* out;
    char ofname[BUF_SIZ];
    char* p;
    strcpy(ofname, fname);
    for (p = ofname + strlen(ofname); p != ofname && *p != '.'; --p);
    if (*p == '.')
        strcpy(p, ext);
    else
        strcat(ofname, ext);
    out = fopen(ofname, "w");
    if (!out) {
        fprintf(stderr, "Could not open output %s\n", ofname);
        exit(-1);
    }
    return out;
}
```

4 Managing output states

The code can be in one of four states: code, default markup, TeX with code quoting, or quiet (the default). In the code state, output is typeset in the environment specified by `envname`, which defaults to the `verbatim` environment. In the default markup state, any comment symbols are stripped from the start of the line, and remainder of each line is sent on to the standard output. In TeX with code quoting, text bracketed by `[[` and `]]` is typeset in teletype font, with any special TeX symbols quoted. In the quiet state, nothing is sent to the output. The `transition` function writes any commands that are needed to transition from one state to the next.

The point of making the quiet state be the default is that there's often a lot of boring boilerplate code (include directives, etc) that aren't really needed by a human reading the code.

```
int transition(FILE* out, int from, int to, const char* envname, int markup)
{
    const char* BV = "begin";
    const char* EV = "end";
    const char* strings[4][4] =
        {{"", EV, EV, EV},
         {BV, "", "", ""},
         {BV, "", "", ""},
         {BV, "", "", ""}};
    if (*strings[from][to]) {
        if (markup == MARKUP_TEX)
            fprintf(out, "\\%s{%s}\n", strings[from][to], envname);
        else if (strings[from][to] == BV)
            fprintf(out, "\n");
        else if (strings[from][to] == EV)
            fprintf(out, "\n");
    }
    return to;
}
```

The `print_clean` line prints a line devoid of extraneous tab characters, since

LaTeX doesn't seem to think that a tab and eight spaces are equivalent (at least, not in `verbatim`).

```
void print_clean(FILE* out, char* s)
{
    for (; s && *s; ++s) {
        if (*s == '\t')
            fprintf(out, "        ");
        else
            fputc(*s, out);
    }
}
```

The `print_clean_quote` line prints a line devoid of extraneous tab characters, and also processes text delimited by `[[` and `]]`.

```
void print_clean_quote(FILE* out, char* s)
{
    static int in_quotes = 0;
    for (; s && *s; ++s) {
        if (s[0] == '[' && s[1] == '[') {
            in_quotes = 1;
            fprintf(out, "{\\tt ");
            for (++s; s[1] == '['; ++s)
                fputc('[', out);
        } else if (s[0] == ']' && s[1] == ']') {
            in_quotes = 0;
            for (++s; s[1] == ']'; ++s)
                fputc(']', out);
            fprintf(out, "}");
        } else if (*s == '\t') {
            fprintf(out, "        ");
        } else if (in_quotes && (*s == '#' || *s == '$' ||
                                 *s == '%' || *s == '&' || *s == '_' ||
                                 *s == '{' || *s == '}')) {
            fputc('\\', out);
            fputc(*s, out);
        } else if (in_quotes && *s == '~') {
            fprintf(out, "\\~{}");
        } else
            fputc(*s, out);
    }
}
```

The `process_line` command takes the current line both in its entirety (`buf`) and with any leading comment characters stripped away (`p`).

```
int process_line(FILE* out, int state, char* p,
                 char* buf, const char* envname, int markup)
{
    if (*p == '@') {
        if (p[1] == 't' || (p[1] == 'T' && markup == MARKUP_MD))
            state = transition(out, state, IN_MARKUP, envname, markup);
        else if (p[1] == 'T')
            state = transition(out, state, IN_TEX_QUOTE, envname, markup);
        else if (p[1] == 'q' || p[1] == 'o')
            state = transition(out, state, IN_QUIET, envname, markup);
        else if (p[1] == 'c')
            state = transition(out, state, IN_CODE, envname, markup);
    } else if (state == IN_CODE) {
        if (markup == MARKUP_MD)
            fprintf(out, "    ");
        print_clean(out, buf);
    } else if (state == IN_MARKUP) {
        print_clean(out, p);
    } else if (state == IN_TEX_QUOTE) {
        print_clean_quote(out, p);
    }
    return state;
}
```

5 Processing simple files

This code works for C, MATLAB, Lua, and shell language files, where a few special characters are used for delimiting comments. FORTRAN, naturally, has to be handled differently. We treat these characters as “skippable” for the purpose of processing markup sections and `@` commands. Note that we're deliberately a little bit sloppy, so that (for example) lines in the middle of a C-style comment block can have a leading asterisk (which will be removed). This same sloppiness would be a problem in shell if we were to allow white space to be skippable – after all, the `@` sign occurs legitimately at the beginning of various make rules. So for shell, the only skippable character will be `#`.

The at-commands to transition between states (`@t` to go to markup, `@T` to go to TEXwith bracket quoting, `@q` to go quiet, or `@c` to go to code) are assumed to be the only useful things on the lines where they occur. These commands can occur after any comment characters, or after asterisks used to mark off the start of a line in the middle of a comment. If the `@` appears elsewhere on the line, it will be ignored.

If Markdown mode is specified, `@T` acts like `@t`.

```
int is_skippable(char c, char* skips)
{
    for (; *skips; ++skips)
        if (c == *skips)
            return 1;
    return 0;
}

void process_simple(FILE* in, FILE* out, char* skips,
                    const char* envname, int markup)
{
    char buf[BUF_SIZ];
    int state = IN_QUIET;
    while (fgets(buf, sizeof(buf), in) != NULL) {
        char* p;
        for (p = buf; is_skippable(*p, skips); ++p);
        state = process_line(out, state, p, buf, envname, markup);
    }
    state = transition(out, state, IN_QUIET, envname, markup);
}
```

6 Processing FORTRAN 77 files

The first six characters of a FORTRAN 77 line are used to indicate comment characters, line continuations, etc. I skip all these.

```
void process_f(FILE* in, FILE* out, const char* envname, int markup)
{
    char buf[BUF_SIZ];
    int state = IN_QUIET;
    while (fgets(buf, sizeof(buf), in) != NULL) {
        int offset = 0;
        while (buf[offset] == '\t' || buf[offset] == ' ' ||
               (offset < 6 && buf[offset] != '\0' &&
                buf[offset] != '\r' && buf[offset] != '\n'))
            ++offset;
        state = process_line(out, state, buf + offset, buf, envname, markup);
    }
    state = transition(out, state, IN_QUIET, envname, markup);
}
```

7 Processing MATLAB batch files

One of the aspects of MATLAB that I find least satisfactory is the requirement that all globally-visible functions be defined in separate files. As a way to get around that, I define the `@o` meta-directive, which defines output redirections. When this processor encounters the `@o` directive at the start of a MATLAB comment line, it will either start, change, or stop the copying of input lines to a second file. For example, if we had `@o foo.m`, subsequent lines would be redirected to the file `foo.m` until the next `@o` or until the EOF. If we had `@o` with no argument, subsequent lines wouldn't be copied until the next `@o` directive. The lines with `@o` directives are not themselves copied to any output.

The MATLAB batch file processing functionality is distinct from the functionality of the other, ordinary language modes, in that it does not produce a LaTeX output file. Also, the MATLAB batch mode can never be selected except by an explicit command-line argument. The reason for this is that I would hate to accidentally clobber an existing m-file prematurely just because I was trying to generate documentation from a preliminary version of the combined files.

```
void process_mb(FILE* in)
{
    char buf[BUF_SIZ];
    FILE* aux_out = NULL;
    while (fgets(buf, sizeof(buf), in) != NULL) {
        char* p;
        for (p = buf; *p == ' ' || *p == '\t' || *p == '%'; ++p);
        if (p[0] == '@' && p[1] == 'o') {
            if (aux_out) {
                fclose(aux_out);
                aux_out = NULL;
            }
            p = strtok(p, " \t\r\n");    /* Skip @o token  */
            p = strtok(NULL, " \t\r\n"); /* Get next token */
            if (p && (aux_out = fopen(p, "w")) == NULL) {
                fprintf(stderr, "Could not open %s for output\n", p);
                exit(-1);
            }
        } else if (aux_out) {
            fputs(buf, aux_out);
        }
    }
    if (aux_out)
        fclose(aux_out);
}
```

In addition to writing out the m-files embedded in a MATLAB batch, we sometimes just want to get a list of what files exist.

```
void list_mb(FILE* in)
{
    char buf[BUF_SIZ];
    while (fgets(buf, sizeof(buf), in) != NULL) {
        char* p;
        for (p = buf; *p == ' ' || *p == '\t' || *p == '%'; ++p);
        if (p[0] == '@' && p[1] == 'o') {
            p = strtok(p, " \t\r\n");    /* Skip @o token  */
            p = strtok(NULL, " \t\r\n"); /* Get next token */
            if (p)
                printf("%s\n", p);
        }
    }
}
```

8 Processing files

For each input file that we process, we have to choose an output file and a language type. The output file we get in one of two ways: if the user specifies a common output file, we use that; otherwise, we try to open a file with the same basename as the original file, but with a different extension. The language type should either be specified explicitly, or it should be DSB_NOLANG (the default); in the latter case, we also pick the language type based on the file extension.

```
void process_file(char* fname, int language, FILE* combined_doc,
                  const char* envname, int markup)
{
    FILE* out = combined_doc;
    FILE* in;
    if (language == DSB_NOLANG)
        language = language_type(fname);

    if ((in = fopen(fname, "r")) == NULL) {
        fprintf(stderr, "Could not open input %s\n", fname);
        exit(-1);
    }
    if (!combined_doc && language != DSB_MB && language != DSB_MB_LIST)
        out = open_markup_output(fname, (markup == MARKUP_TEX ?
                                         ".tex" : ".md"));


    if (language == DSB_C)
        process_simple(in, out, " \t/*", envname, markup);
    else if (language == DSB_M)
        process_simple(in, out, " \t%", envname, markup);
    else if (language == DSB_MB)
        process_mb(in);
    else if (language == DSB_MB_LIST)
        list_mb(in);
    else if (language == DSB_F77)
        process_f(in, out, envname, markup);
    else if (language == DSB_LUA)
        process_simple(in, out, " \t-", envname, markup);
    else if (language == DSB_SH)
        process_simple(in, out, "#", envname, markup);
    else if (language == DSB_LISP)
        process_simple(in, out, " \t;", envname, markup);

    if (!combined_doc && out)
        fclose(out);
    fclose(in);
}
```

9 Document header

If you want to define lots of fancy macros and combine multiple LaTeX files into a single document, then probably you should not be using the simple LaTeX produced by the `-d` option. However, because `-d` is so handy for debugging the LaTeX in a single file, and because I write many LaTeX files that require at least a few extra packages (e.g. `amsmath` or `amssymb`), the processor lets you specify a file to be included in the preamble via an input command command. You can specify the preamble file by the environment variable `DSBWEB_TEX` or by the `-p` flag.

```
#define PREAMBLE_VAR "DSBWEB_TEX"

void write_preamble(FILE* out, char* preamble)
{
    if (!preamble)
        preamble = getenv(PREAMBLE_VAR);

    fprintf(out, "\\documentclass{article}\n");
    if (preamble && *preamble)
        fprintf(out, "\\input{%s}\n", preamble);
    fprintf(out, "\n\\begin{document}\n");
}
```

10 Main routine

The routine takes four types of arguments:

1. `-md` is used to specify Markdown mode, as opposed to the default LaTeX mode.
2. `-o [fname]` or `-d [fname]` flags that specify a common output file and `-p [fname]` flags that specify a file to include in the TeX preamble. All the files that are processed will go to that same common output; there can be only one. If the `-d` flag is used, the common output file will be a complete LaTeX document, including a preamble and begin/end document.
3. `-c`, `-m`, `-mb`, `-f`, `-l`, or `-sh` flags that specify that subsequent files should be processed as C, MATLAB, MATLAB batches, FORTRAN, Lua, or shell files. The `-list` flag specifies that we want the names of any files written out from a MATLAB batch to go to be written out the screen.
4. Input file names.

The main routine just processes all the arguments in an appropriate order (`-o`, `-d`, and `-p` flags first, then all the language flags and file names in the order specified).

```
int main(int argc, char** argv)
{
    int i;
    int full_doc      = 0;          /* Is this a full LaTeX document?   */
    FILE* out         = NULL;       /* Pointer for a common output file */
    int language_flag = DSB_NOLANG; /* Current specified langauge       */
    char* preamble    = NULL;       /* File to include in TeX preamble  */
    char* envname     = "verbatim"; /* TeX environment for code         */
    int markup        = MARKUP_TEX; /* Default to TeX for markup        */

    for (i = 1; i < argc; ++i) {

        /*
         * Process help directives
         */
        if (strcmp(argv[i], "-help") == 0) {
            fprintf(stderr, DSBWEB_HELP);
            return 0;
        }

        /*
         * Check for Markdown mode specifier
         */
        if (strcmp(argv[i], "-md") == 0) {
            markup = MARKUP_MD;
            continue;
        }
        /*
         * Check for -d, -o, -p, or -e output specifications
         */
        if (strcmp(argv[i], "-d") == 0 ||
            strcmp(argv[i], "-o") == 0 ||
            strcmp(argv[i], "-p") == 0 ||
            strcmp(argv[i], "-e") == 0) {

            if (i+1 == argc) {
                fprintf(stderr, "Flag %s requires an argument\n", argv[i]);
                exit(-1);
            }

            /* Handle preamble directives */
            if (argv[i][1] == 'p') {
                preamble = argv[i+1];
                continue;
            }

            /* Handle environment name directives */
            if (argv[i][1] == 'e') {
                envname = argv[i+1];
                continue;
            }

            /* Handle -d and -o */
            full_doc = (argv[i][1] == 'd');
            if (out != NULL) {
                fprintf(stderr, "Cannot specify multiple explicit outputs\n");
                exit(-1);
            } else if ((out = fopen(argv[i+1], "w")) == NULL) {
                fprintf(stderr, "Could not open output %s\n", argv[i+1]);
                exit(-1);
            }
        }
    }

    if (full_doc && markup == MARKUP_TEX)
        write_preamble(out, preamble);

    /*
     * Process files and language specification flags
     */
    for (--argc, ++argv; argc > 0; --argc, ++argv) {
        if (strcmp(argv[0], "-c") == 0) {
            language_flag = DSB_C;
        } else if (strcmp(argv[0], "-m") == 0) {
            language_flag = DSB_M;
        } else if (strcmp(argv[0], "-mb") == 0) {
            language_flag = DSB_MB;
        } else if (strcmp(argv[0], "-f") == 0) {
            language_flag = DSB_F77;
        } else if (strcmp(argv[0], "-l") == 0) {
            language_flag = DSB_LUA;
        } else if (strcmp(argv[0], "-sh") == 0) {
            language_flag = DSB_SH;
        } else if (strcmp(argv[0], "-list") == 0) {
            language_flag = DSB_MB_LIST;
        } else if (strcmp(argv[0], "-d") == 0 ||
                   strcmp(argv[0], "-o") == 0 ||
                   strcmp(argv[0], "-p") == 0 ||
                   strcmp(argv[0], "-e") == 0) {
            --argc, ++argv;
        } else if (strcmp(argv[0], "-md") == 0) {
        } else if (argv[0][0] == '-') {
            fprintf(stderr, "Unrecognized flag: %s\n", argv[0]);
            return -1;
        } else {
            process_file(argv[0], language_flag, out, envname, markup);
        }
    }

    if (full_doc && markup == MARKUP_TEX)
        fprintf(out, "\\end{document}\n");
    if (out)
        fclose(out);

    return 0;
}
```