# Common code for CTANGLE and CWEAVE

(Version 4.7 [TEX Live])






March 21, 2022 at 18:48

**1* Introduction.** This file contains code common to `CTANGLE`, `CWEAVE`, and `CTWILL`, which roughly concerns the following problems: character uniformity, input routines, error handling and parsing of command line. We have tried to concentrate in this file all the system dependencies, so as to maximize portability.

In the texts below we will sometimes use `CWEB` to refer to any of the three component programs, if no confusion can arise.

The file begins with a few basic definitions.

⟨ Include files 3* ⟩
⟨ Preprocessor definitions ⟩
⟨ Common code for `CWEAVE` and `CTANGLE` 2* ⟩
⟨ Global variables 18* ⟩
⟨ Predeclaration of procedures 7* ⟩

**2*** The details will be filled in due course. The interface of this module is included first. It is also used by the main programs.

First comes general stuff:

⟨ Common code for `CWEAVE` and `CTANGLE` 2* ⟩ ≡

```
typedef uint8_t eight_bits;
typedef uint16_t sixteen_bits;
typedef enum {
  ctangle, cweave, ctwill
} cweb;
extern cweb program;     ▷ CTANGLE or CWEAVE or CTWILL? ◁
extern int phase;     ▷ which phase are we in? ◁
```

See also sections 4*, 5*, 6*, 8*, 9*, 11*, 13*, and 14*.

This code is used in section 1*.

**3*** You may have noticed that almost all `"strings"` in the `CWEB` sources are placed in the context of the '_' macro. This is just a shortcut for the '*gettext*' function from the "GNU gettext utilities." For systems that do not have this library installed, we wrap things for neutral behavior without internationalization. For backward compatibility with pre-ANSI compilers, we replace the "standard" header file '`stdbool.h`' with the KPATHSEA interface '`simpletypes.h`'.

```
#define _(s) gettext(s)
```

⟨ Include files 3* ⟩ ≡

```
#include <ctype.h>     ▷ definition of isalpha, isdigit and so on ◁
#include <kpathsea/simpletypes.h>     ▷ boolean, true and false ◁
#include <stddef.h>     ▷ definition of ptrdiff_t ◁
#include <stdint.h>     ▷ definition of uint8_t and uint16_t ◁
#include <stdio.h>     ▷ definition of printf and friends ◁
#include <stdlib.h>     ▷ definition of getenv and exit ◁
#include <string.h>     ▷ definition of strlen, strcmp and so on ◁
#ifndef HAVE_GETTEXT
#define HAVE_GETTEXT 0
#endif
#if HAVE_GETTEXT
#include <libintl.h>
#else
#define gettext(a) a
#endif
```

See also sections 91*, 93*, and 96*.

This code is used in section 1*.

**4*** Code related to the character set:

#**define** *and_and* °*4* ▷ '&&' ; corresponds to MIT's ∧ ◁
#**define** *lt_lt* °*20* ▷ '<<' ; corresponds to MIT's ⊂ ◁
#**define** *gt_gt* °*21* ▷ '>>' ; corresponds to MIT's ⊃ ◁
#**define** *plus_plus* °*13* ▷ '++' ; corresponds to MIT's ↑ ◁
#**define** *minus_minus* °*1* ▷ '--' ; corresponds to MIT's ↓ ◁
#**define** *minus_gt* °*31* ▷ '->' ; corresponds to MIT's → ◁
#**define** *non_eq* °*32* ▷ '!=' ; corresponds to MIT's ≠ ◁
#**define** *lt_eq* °*34* ▷ '<=' ; corresponds to MIT's ≤ ◁
#**define** *gt_eq* °*35* ▷ '>=' ; corresponds to MIT's ≥ ◁
#**define** *eq_eq* °*36* ▷ '==' ; corresponds to MIT's ≡ ◁
#**define** *or_or* °*37* ▷ '||' ; corresponds to MIT's ∨ ◁
#**define** *dot_dot_dot* °*16* ▷ '...' ; corresponds to MIT's ϖ ◁
#**define** *colon_colon* °*6* ▷ '::' ; corresponds to MIT's ∈ ◁
#**define** *period_ast* °*26* ▷ '.*' ; corresponds to MIT's ⊗ ◁
#**define** *minus_gt_ast* °*27* ▷ '->*' ; corresponds to MIT's ↰ ◁

#**define** *compress*(*c*) **if** (*loc*++ ≤ *limit*) **return** *c*

⟨ Common code for `CWEAVE` and `CTANGLE` 2* ⟩ +≡
**extern char** *section_text*[ ]; ▷ text being sought for ◁
**extern char** ∗*section_text_end*; ▷ end of *section_text* ◁
**extern char** ∗*id_first*; ▷ where the current identifier begins in the buffer ◁
**extern char** ∗*id_loc*; ▷ just after the current identifier in the buffer ◁

**5*** Code related to input routines:

#**define** *xisalpha*(*c*) (*isalpha*((**int**)(*c*)) ∧ ((**eight_bits**)(*c*) < °*200*))
#**define** *xisdigit*(*c*) (*isdigit*((**int**)(*c*)) ∧ ((**eight_bits**)(*c*) < °*200*))
#**define** *xisspace*(*c*) (*isspace*((**int**)(*c*)) ∧ ((**eight_bits**)(*c*) < °*200*))
#**define** *xislower*(*c*) (*islower*((**int**)(*c*)) ∧ ((**eight_bits**)(*c*) < °*200*))
#**define** *xisupper*(*c*) (*isupper*((**int**)(*c*)) ∧ ((**eight_bits**)(*c*) < °*200*))
#**define** *xisxdigit*(*c*) (*isxdigit*((**int**)(*c*)) ∧ ((**eight_bits**)(*c*) < °*200*))
#**define** *isxalpha*(*c*) ((*c*) ≡ '_' ∨ (*c*) ≡ '$') ▷ non-alpha characters allowed in identifier ◁
#**define** *ishigh*(*c*) ((**eight_bits**)(*c*) > °*177*)

⟨ Common code for `CWEAVE` and `CTANGLE` 2* ⟩ +≡
**extern char** *buffer*[ ]; ▷ where each line of input goes ◁
**extern char** ∗*buffer_end*; ▷ end of *buffer* ◁
**extern char** ∗*loc*; ▷ points to the next character to be read from the buffer ◁
**extern char** ∗*limit*; ▷ points to the last character in the buffer ◁

**6*.** Code related to file handling:

```
format line x ▷ make line an unreserved word ◁
#define max_include_depth 10
        ▷ maximum number of source files open simultaneously, not counting the change file ◁
#define max_file_name_length 1024
#define cur_file file[include_depth] ▷ current file ◁
#define cur_file_name file_name[include_depth] ▷ current file name ◁
#define cur_line line[include_depth] ▷ number of current line in current file ◁
#define web_file file[0] ▷ main source file ◁
#define web_file_name file_name[0] ▷ main source file name ◁
```

⟨ Common code for CWEAVE and CTANGLE 2* ⟩ +≡

```
extern int include_depth; ▷ current level of nesting ◁
extern FILE *file[]; ▷ stack of non-change files ◁
extern FILE *change_file; ▷ change file ◁
extern char file_name[][max_file_name_length]; ▷ stack of non-change file names ◁
extern char change_file_name[]; ▷ name of change file ◁
extern char check_file_name[]; ▷ name of check_file ◁
extern int line[]; ▷ number of current line in the stacked files ◁
extern int change_line; ▷ number of current line in change file ◁
extern int change_depth; ▷ where @y originated during a change ◁
extern boolean input_has_ended; ▷ if there is no more input ◁
extern boolean changing; ▷ if the current line is from change_file ◁
extern boolean web_file_open; ▷ if the web file is being read ◁
```

**7*.** ⟨ Predeclaration of procedures 7* ⟩ ≡

```
extern boolean get_line(void); ▷ inputs the next line ◁
extern void check_complete(void); ▷ checks that all changes were picked up ◁
extern void reset_input(void); ▷ initialize to read the web file and change file ◁
```

See also sections 10*, 12*, 15*, 24, 28, 33, 55, 64, 76, and 98*.

This code is used in section 1*.

**8*.** Code related to section numbers:

⟨ Common code for CWEAVE and CTANGLE 2* ⟩ +≡

```
extern sixteen_bits section_count; ▷ the current section number ◁
extern boolean changed_section[]; ▷ is the section changed? ◁
extern boolean change_pending; ▷ is a decision about change still unclear? ◁
extern boolean print_where; ▷ tells CTANGLE to print line and file info ◁
```

**9*** Code related to identifier and section name storage:

```
#define length(c) (size_t)((c+1)→byte_start − (c)→byte_start)  ▷ the length of a name ◁
#define print_id(c) term_write((c)→byte_start, length(c))  ▷ print identifier ◁
#define llink link  ▷ left link in binary search tree for section names ◁
#define rlink dummy.Rlink  ▷ right link in binary search tree for section names ◁
#define root name_dir→rlink  ▷ the root of the binary search tree for section names ◁
⟨ Common code for CWEAVE and CTANGLE 2* ⟩ +≡
  typedef struct name_info {
    char *byte_start;  ▷ beginning of the name in byte_mem ◁
    struct name_info *link;
    union {
      struct name_info *Rlink;  ▷ right link in binary search tree for section names ◁
      char Ilk;  ▷ used by identifiers in CWEAVE only ◁
    } dummy;
    void *equiv_or_xref;  ▷ info corresponding to names ◁
  } name_info;  ▷ contains information about an identifier or section name ◁
  typedef name_info *name_pointer;  ▷ pointer into array of name_infos ◁
  typedef name_pointer *hash_pointer;
  extern char byte_mem[];  ▷ characters of names ◁
  extern char *byte_mem_end;  ▷ end of byte_mem ◁
  extern char *byte_ptr;  ▷ first unused position in byte_mem ◁
  extern name_info name_dir[];  ▷ information about names ◁
  extern name_pointer name_dir_end;  ▷ end of name_dir ◁
  extern name_pointer name_ptr;  ▷ first unused position in name_dir ◁
  extern name_pointer hash[];  ▷ heads of hash lists ◁
  extern hash_pointer hash_end;  ▷ end of hash ◁
  extern hash_pointer h;  ▷ index into hash-head array ◁
```

**10*** ⟨ Predeclaration of procedures 7* ⟩ +≡

```
  extern boolean names_match(name_pointer, const char *, size_t, eight_bits);
  extern name_pointer id_lookup(const char *, const char *, eight_bits);
    ▷ looks up a string in the identifier table ◁
  extern name_pointer section_lookup(char *, char *, boolean);  ▷ finds section name ◁
  extern void init_node(name_pointer);
  extern void init_p(name_pointer, eight_bits);
  extern void print_prefix_name(name_pointer);
  extern void print_section_name(name_pointer);
  extern void sprint_section_name(char *, name_pointer);
```

**11*** Code related to error handling:

```
#define spotless 0  ▷ history value for normal jobs ◁
#define harmless_message 1  ▷ history value when non-serious info was printed ◁
#define error_message 2  ▷ history value when an error was noted ◁
#define fatal_message 3  ▷ history value when we had to stop prematurely ◁
#define mark_harmless if (history ≡ spotless) history ← harmless_message
#define mark_error history ← error_message
#define confusion(s) fatal(_("! This can't happen: "), s)
⟨ Common code for CWEAVE and CTANGLE 2* ⟩ +≡
  extern int history;  ▷ indicates how bad this run was ◁
```

**12.*** ⟨ Predeclaration of procedures 7* ⟩ +≡

```
extern int wrap_up(void);  ▷ indicate history and exit ◁
extern void err_print(const char *);  ▷ print error message and context ◁
extern void fatal(const char *, const char *);  ▷ issue error message and die ◁
extern void overflow(const char *);  ▷ succumb because a table has overflowed ◁
```

**13.*** Code related to command line arguments:

```
#define show_banner flags['b']  ▷ should the banner line be printed? ◁
#define show_progress flags['p']  ▷ should progress reports be printed? ◁
#define show_happiness flags['h']  ▷ should lack of errors be announced? ◁
#define show_stats flags['s']  ▷ should statistics be printed at end of run? ◁
#define make_xrefs flags['x']  ▷ should cross references be output? ◁
#define check_for_change flags['c']  ▷ check temporary output for changes ◁
```

⟨ Common code for CWEAVE and CTANGLE 2* ⟩ +≡

```
extern int argc;  ▷ copy of ac parameter to main ◁
extern char **argv;  ▷ copy of av parameter to main ◁
extern char C_file_name[];  ▷ name of C_file ◁
extern char tex_file_name[];  ▷ name of tex_file ◁
extern char idx_file_name[];  ▷ name of idx_file ◁
extern char scn_file_name[];  ▷ name of scn_file ◁
extern boolean flags[];  ▷ an option for each 7-bit code ◁
extern const char *use_language;  ▷ prefix to cwebmac.tex in TEX output ◁
```

**14.*** Code related to output:

```
#define update_terminal fflush(stdout)  ▷ empty the terminal output buffer ◁
#define new_line putchar('\n')
#define term_write(a,b) fflush(stdout), fwrite(a, sizeof(char), b, stdout)
```

⟨ Common code for CWEAVE and CTANGLE 2* ⟩ +≡

```
extern FILE *C_file;  ▷ where output of CTANGLE goes ◁
extern FILE *tex_file;  ▷ where output of CWEAVE goes ◁
extern FILE *idx_file;  ▷ where index from CWEAVE goes ◁
extern FILE *scn_file;  ▷ where list of sections from CWEAVE goes ◁
extern FILE *active_file;  ▷ currently active file for CWEAVE output ◁
extern FILE *check_file;  ▷ temporary output file ◁
```

**15.*** The procedure that gets everything rolling:

⟨ Predeclaration of procedures 7* ⟩ +≡

```
extern void common_init(void);
extern void print_stats(void);
extern void cb_show_banner(void);
```

**16.*** The following parameters are sufficient to handle TEX (converted to CWEB), so they should be sufficient for most applications of CWEB.

```
#define buf_size 1000  ▷ maximum length of input line, plus one ◁
#define longest_name 10000  ▷ file names, section names, and section texts shouldn't be longer than this ◁
#define long_buf_size (buf_size + longest_name)  ▷ for CWEAVE ◁
#define max_bytes 1000000
        ▷ the number of bytes in identifiers, index entries, and section names; must be less than 2^24 ◁
#define max_names 10239  ▷ number of identifiers, strings, section names; must be less than 10240 ◁
#define max_sections 4000  ▷ greater than the total number of sections ◁
```

**17.*** End of `COMMON` interface.

**18.*** In certain cases `CTANGLE` and `CWEAVE` should do almost, but not quite, the same thing. In these cases we've written common code for both, differentiating between the two by means of the global variable *program*. And `CTWILL` adds some extra twists.

```
⟨ Global variables 18* ⟩ ≡
  cweb program;      ▷ CTANGLE or CWEAVE or CTWILL? ◁
```

See also sections 19, 21, 22, 25*, 26, 37, 43, 44, 46*, 65, 73*, 83*, 86*, and 87*.

This code is used in section 1*.

**20.*** There's an initialization procedure that gets both `CTANGLE` and `CWEAVE` off to a good start. We will fill in the details of this procedure later.

```
void common_init(void)
{
  ⟨ Initialize pointers 45 ⟩
  ⟨ Set up PROGNAME feature and initialize the search path mechanism 94* ⟩
  ⟨ Set locale and bind language catalogs 92* ⟩
  ⟨ Set the default options common to CTANGLE and CWEAVE 74* ⟩
  ⟨ Scan arguments and open output files 84* ⟩
}
```

**23.*** In the unlikely event that your standard I/O library does not support *feof*, *getc*, and *ungetc* you may have to change things here.

```
static boolean input_ln(      ▷ copies a line into buffer or returns false ◁
    FILE *fp)      ▷ what file to read from ◁
{
  register int c ← EOF;      ▷ character read; initialized so some compilers won't complain ◁
  register char *k;      ▷ where next character goes ◁
  if (feof(fp)) return false;      ▷ we have hit end-of-file ◁
  limit ← k ← buffer;      ▷ beginning of buffer ◁
  while (k ≤ buffer_end ∧ (c ← getc(fp)) ≠ EOF ∧ c ≠ '\n')
    if ((*(k++) ← c) ≠ ' ' ∧ c ≠ '\r') limit ← k;
  if (k > buffer_end)
    if ((c ← getc(fp)) ≠ EOF ∧ c ≠ '\n') {
      ungetc(c, fp); loc ← buffer; err_print(_("! Input line too long"));
    }
  if (c ≡ EOF ∧ limit ≡ buffer) return false;      ▷ there was nothing after the last newline ◁
  return true;
}
```

**25.*** Now comes the problem of deciding which file to read from next. Recall that the actual text that CWEB should process comes from two streams: a *web_file*, which can contain possibly nested include commands `@i`, and a *change_file*, which might also contain includes. The *web_file* together with the currently open include files form a stack *file*, whose names are stored in a parallel stack *file_name*. The boolean *changing* tells whether or not we're reading from the *change_file*.

The line number of each open file is also kept for error reporting and for the benefit of CTANGLE.

```
⟨ Global variables 18* ⟩ +≡
  int include_depth; ▷ current level of nesting ◁
  FILE *file[max_include_depth]; ▷ stack of non-change files ◁
  FILE *change_file; ▷ change file ◁
  char file_name[max_include_depth][max_file_name_length]; ▷ stack of non-change file names ◁
  char change_file_name[max_file_name_length]; ▷ name of change file ◁
  int line[max_include_depth]; ▷ number of current line in the stacked files ◁
  int change_line; ▷ number of current line in change file ◁
  int change_depth; ▷ where @y originated during a change ◁
  boolean input_has_ended; ▷ if there is no more input ◁
  boolean changing; ▷ if the current line is from change_file ◁
  boolean web_file_open ← false; ▷ if the web file is being read ◁
```

**29.*** While looking for a line that begins with `@x` in the change file, we allow lines that begin with `@`, as long as they don't begin with `@y`, `@z`, or `@i` (which would probably mean that the change file is fouled up).

```
⟨ Skip over comment lines in the change file; return if end of file 29* ⟩ ≡
  while (true) {
    change_line++;
    if (¬input_ln(change_file)) return;
    if (limit < buffer + 2) continue;
    if (buffer[0] ≠ '@') continue;
    if (xisupper(buffer[1])) buffer[1] ← tolower((int) buffer[1]);
    if (buffer[1] ≡ 'x') break;
    if (buffer[1] ≡ 'y' ∨ buffer[1] ≡ 'z' ∨ buffer[1] ≡ 'i') {
      loc ← buffer + 2; err_print(_("! Missing @x in change file"));
    }
  }
```

This code is used in section 27.

**30.*** Here we are looking at lines following the `@x`.

```
⟨ Skip to the next nonblank line; return if end of file 30* ⟩ ≡
  do {
    change_line++;
    if (¬input_ln(change_file)) {
      err_print(_("! Change file ended after @x")); return;
    }
  } while (limit ≡ buffer);
```

This code is used in section 27.

**32.*** The following procedure is used to see if the next change entry should go into effect; it is called only when *changing* is *false*. The idea is to test whether or not the current contents of *buffer* matches the current contents of *change_buffer*. If not, there's nothing more to do; but if so, a change is called for: All of the text down to the `@y` is supposed to match. An error message is issued if any discrepancy is found. Then the procedure prepares to read the next line from *change_file*.

When a match is found, the current section is marked as changed unless the first line after the `@x` and after the `@y` both start with either '`@*`' or '`@␣`' (possibly preceded by whitespace).

This procedure is called only when *buffer* < *limit*, i.e., when the current line is nonempty.

```
#define if_section_start_make_pending(b)
        *limit ← '!'; for (loc ← buffer; xisspace(*loc); loc++) ; *limit ← '␣';
        if (*loc ≡ '@' ∧ (xisspace(*(loc + 1)) ∨ *(loc + 1) ≡ '*')) change_pending ← b
  static void check_change(void)      ▷ switches to change_file if the buffers match ◁
  {
    int n ← 0;      ▷ the number of discrepancies found ◁
    if (lines_dont_match) return;
    change_pending ← false;
    if (¬changed_section[section_count]) {
      if_section_start_make_pending(true);
      if (¬change_pending) changed_section[section_count] ← true;
    }
    while (true) {
      changing ← print_where ← true; change_line++;
      if (¬input_ln(change_file)) {
        err_print(_("!␣Change␣file␣ended␣before␣@y")); change_limit ← change_buffer;
        changing ← false; return;
      }
      if (limit > buffer + 1 ∧ buffer[0] ≡ '@') {
        char xyz_code ← xisupper(buffer[1]) ? tolower((int) buffer[1]) : buffer[1];
        ⟨ If the current line starts with @y, report any discrepancies and return 34* ⟩
      }
      ⟨ Move buffer and limit to change_buffer and change_limit 31 ⟩
      changing ← false; cur_line++;
      while (¬input_ln(cur_file)) {      ▷ pop the stack or quit ◁
        if (include_depth ≡ 0) {
          err_print(_("!␣CWEB␣file␣ended␣during␣a␣change")); input_has_ended ← true; return;
        }
        include_depth--; cur_line++;
      }
      if (lines_dont_match) n++;
    }
  }
```

**34.*** ⟨ If the current line starts with `@y`, report any discrepancies and **return** 34* ⟩ ≡

```
if (xyz_code ≡ 'x' ∨ xyz_code ≡ 'z') {
  loc ← buffer + 2; err_print(_("! Where is the matching @y?"));
}
else if (xyz_code ≡ 'y') {
  if (n > 0) {
    loc ← buffer + 2; printf("\n! Hmm... %d ", n);
    err_print(_("of the preceding lines failed to match"));
  }
  change_depth ← include_depth; return;
}
```

This code is used in section 32*.

**36.*** The following code opens the input files.

⟨ Open input files 36* ⟩ ≡

```
if ((found_filename ← kpse_find_cweb(web_file_name)) ≡ Λ
      ∨ (web_file ← fopen(found_filename, "r")) ≡ Λ)
  fatal(_("! Cannot open input file "), web_file_name);
else if (strlen(found_filename) < max_file_name_length) {      ▷ Copy name for #line directives. ◁
  if (strcmp(web_file_name, found_filename))
    strcpy(web_file_name, found_filename + ((strncmp(found_filename, "./", 2) ≡ 0) ? 2 : 0));
  free(found_filename);
}
else fatal(_("! Filename too long\n"), found_filename);
web_file_open ← true;
if ((found_filename ← kpse_find_cweb(change_file_name)) ≡ Λ
      ∨ (change_file ← fopen(found_filename, "r")) ≡ Λ)
  fatal(_("! Cannot open change file "), change_file_name);
else if (strlen(found_filename) < max_file_name_length) {      ▷ Copy name for #line directives. ◁
  if (strcmp(change_file_name, found_filename))
    strcpy(change_file_name, found_filename + ((strncmp(found_filename, "./", 2) ≡ 0) ? 2 : 0));
  free(found_filename);
}
else fatal(_("! Filename too long\n"), found_filename);
```

This code is used in section 35.

**38.*** **boolean** *get_line*(**void**) ▷ inputs the next line ◁

```
{
restart:
  if (changing ∧ include_depth ≡ change_depth)
    ⟨ Read from change_file and maybe turn off changing 41* ⟩
  if (¬changing ∨ include_depth > change_depth) {
    ⟨ Read from cur_file and maybe turn on changing 40 ⟩
    if (changing ∧ include_depth ≡ change_depth) goto restart;
  }
  if (input_has_ended) return false;
  loc ← buffer; *limit ← '␣';
  if (buffer[0] ≡ '@' ∧ (buffer[1] ≡ 'i' ∨ buffer[1] ≡ 'I')) {
    loc ← buffer + 2; *limit ← '"';
    while (*loc ≡ '␣' ∨ *loc ≡ '\t') loc++;
    if (loc ≥ limit) {
      err_print(_("!␣Include␣file␣name␣not␣given")); goto restart;
    }
    if (include_depth ≥ max_include_depth − 1) {
      err_print(_("!␣Too␣many␣nested␣includes")); goto restart;
    }
    include_depth++;      ▷ push input stack ◁
    ⟨ Try to open include file, abort push if unsuccessful, go to restart 39* ⟩
  }
  return true;
}
```

**39:*** When an `@i` line is found in the *cur_file*, we must temporarily stop reading it and start reading from the named include file. The `@i` line should give a complete file name with or without double quotes. The actual file lookup is done with the help of the KPATHSEA library; see section ⟨ File lookup with KPATHSEA 93 ⟩ for details. The remainder of the `@i` line after the file name is ignored.

```
#define too_long()
        {
          include_depth--; err_print(_("! Include file name too long")); goto restart;
        }
```

⟨ Try to open include file, abort push if unsuccessful, go to *restart* 39* ⟩ ≡

```
{
  char *cur_file_name_end ← cur_file_name + max_file_name_length − 1;
  char *k ← cur_file_name;
  if (*loc ≡ '"') {
    loc++;
    while (*loc ≠ '"' ∧ k ≤ cur_file_name_end) *k++ ← *loc++;
    if (loc ≡ limit) k ← cur_file_name_end + 1;      ▷ unmatched quote is 'too long' ◁
  }
  else
    while (*loc ≠ ' ' ∧ *loc ≠ '\t' ∧ *loc ≠ '"' ∧ k ≤ cur_file_name_end) *k++ ← *loc++;
  if (k > cur_file_name_end) too_long();
  *k ← '\0';
  if ((found_filename ← kpse_find_cweb(cur_file_name)) ≠ Λ
      ∧ (cur_file ← fopen(found_filename, "r")) ≠ Λ) {      ▷ Copy name for #line directives. ◁
    if (strlen(found_filename) < max_file_name_length) {
      if (strcmp(cur_file_name, found_filename))
        strcpy(cur_file_name, found_filename + ((strncmp(found_filename, "./", 2) ≡ 0) ? 2 : 0));
      free(found_filename);
    }
    else fatal(_("! Filename too long\n"), found_filename);
    cur_line ← 0; print_where ← true; goto restart;      ▷ success ◁
  }
  include_depth--; err_print(_("! Cannot open include file")); goto restart;
}
```

This code is used in section 38*.

**41*.** ⟨ Read from *change_file* and maybe turn off *changing* 41* ⟩ ≡

```
{
  change_line++;
  if (¬input_ln(change_file)) {
    err_print(_("!␣Change␣file␣ended␣without␣@z"));  buffer[0] ← '@';  buffer[1] ← 'z';
    limit ← buffer + 2;
  }
  if (limit > buffer) {      ▷ check if the change has ended ◁
    if (change_pending) {
      if_section_start_make_pending(false);
      if (change_pending) {
        changed_section[section_count] ← true;  change_pending ← false;
      }
    }
    *limit ← '␣';
    if (buffer[0] ≡ '@') {
      if (xisupper(buffer[1]))  buffer[1] ← tolower((int) buffer[1]);
      if (buffer[1] ≡ 'x' ∨ buffer[1] ≡ 'y') {
        loc ← buffer + 2;  err_print(_("!␣Where␣is␣the␣matching␣@z?"));
      }
      else if (buffer[1] ≡ 'z') {
        prime_the_change_buffer( );  changing ← ¬changing;  print_where ← true;
      }
    }
  }
}
```

This code is used in section 38*.

**42*.** At the end of the program, we will tell the user if the change file had a line that didn't match any relevant line in *web_file*.

```
void check_complete(void)
{
  if (change_limit ≠ change_buffer) {      ▷ changing is false ◁
    strncpy(buffer, change_buffer, (size_t)(change_limit − change_buffer + 1));
    limit ← buffer + (ptrdiff_t)(change_limit − change_buffer);  changing ← true;
    change_depth ← include_depth;  loc ← buffer;
    err_print(_("!␣Change␣file␣entry␣did␣not␣match"));
  }
}
```

**46*.** The hash table itself consists of *hash_size* entries of type **name_pointer**, and is updated by the *id_lookup* procedure, which finds a given identifier and returns the appropriate **name_pointer**. The matching is done by the function *names_match*, which is slightly different in `CWEAVE` and `CTANGLE`. If there is no match for the identifier, it is inserted into the table.

```
#define hash_size 8501      ▷ should be prime ◁
⟨ Global variables 18* ⟩ +≡
  name_pointer hash[hash_size];      ▷ heads of hash lists ◁
  hash_pointer hash_end ← hash + hash_size − 1;      ▷ end of hash ◁
  hash_pointer h;      ▷ index into hash-head array ◁
```

**51:*** The information associated with a new identifier must be initialized in a slightly different way in `CWEAVE` than in `CTANGLE`; hence the *init_p* procedure.

```
⟨ Enter a new name into the table at position p 51* ⟩ ≡
  {
    if (byte_ptr + l > byte_mem_end) overflow(_("byte␣memory"));
    if (name_ptr ≥ name_dir_end) overflow(_("name"));
    strncpy(byte_ptr, first, l); (++name_ptr)→byte_start ← byte_ptr += l; init_p(p, t);
  }
```

This code is used in section 48.

**57:*** Adding a section name to the tree is straightforward if we know its parent and whether it's the *rlink* or *llink* of the parent. As a special case, when the name is the first section being added, we set the "parent" to Λ. When a section name is created, it has only one chunk, which however may be just a prefix; the full name will hopefully be unveiled later. Obviously, *prefix_length* starts out as the length of the first chunk, though it may decrease later.

The information associated with a new node must be initialized differently in `CWEAVE` and `CTANGLE`; hence the *init_node* procedure, which is defined differently in `cweave.w` and `ctangle.w`.

```
static name_pointer add_section_name(      ▷ install a new node in the tree ◁
    name_pointer par,      ▷ parent of new node ◁
    int c,      ▷ right or left? ◁
    char *first,      ▷ first character of section name ◁
    char *last,      ▷ last character of section name, plus one ◁
    boolean ispref)      ▷ are we adding a prefix or a full name? ◁
{
  name_pointer p ← name_ptr;      ▷ new node ◁
  char *s ← first_chunk(p);
  size_t name_len ← (size_t)(last − first + (int) ispref);      ▷ length of section name ◁
  if (s + name_len > byte_mem_end) overflow(_("byte␣memory"));
  if (name_ptr + 1 ≥ name_dir_end) overflow(_("name"));
  (++name_ptr)→byte_start ← byte_ptr ← s + name_len;
  if (ispref) {
    *(byte_ptr − 1) ← '␣'; name_len−−; name_ptr→link ← name_dir;
    (++name_ptr)→byte_start ← byte_ptr;
  }
  set_prefix_length(p, name_len); strncpy(s, first, name_len); p→llink ← p→rlink ← Λ; init_node(p);
  return par ≡ Λ ? (root ← p) : c ≡ less ? (par→llink ← p) : (par→rlink ← p);
}
```

**58.*** **static void** *extend_section_name*(**name_pointer** *p*, ▷ name to be extended ◁
**char** ∗*first*, ▷ beginning of extension text ◁
**char** ∗*last*, ▷ one beyond end of extension text ◁
**boolean** *ispref*) ▷ are we adding a prefix or a full name? ◁

```
{
  char *s;
  name_pointer q ← p + 1;
  size_t name_len ← (size_t)(last − first + (int) ispref);
  if (name_ptr ≥ name_dir_end) overflow(_("name"));
  while (q→link ≠ name_dir) q ← q→link;
  q→link ← name_ptr; s ← name_ptr→byte_start; name_ptr→link ← name_dir;
  if (s + name_len > byte_mem_end) overflow(_("byte␣memory"));
  (++name_ptr)→byte_start ← byte_ptr ← s + name_len; strncpy(s, first, name_len);
  if (ispref) *(byte_ptr − 1) ← '␣';
}
```

**60.*** A legal new name matches an existing section name if and only if it matches the shortest prefix of that section name. Therefore we can limit our search for matches to shortest prefixes, which eliminates the need for chunk-chasing at this stage.

⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 60* ⟩ ≡

```
while (p) {  ▷ compare shortest prefix of p with new name ◁
  c ← web_strcmp(first, name_len, first_chunk(p), prefix_length(p));
  if (c ≡ less ∨ c ≡ greater) {  ▷ new name does not match p ◁
    if (r ≡ Λ)  ▷ no previous matches have been found ◁
      par ← p;
    p ← (c ≡ less ? p→llink : p→rlink);
  }
  else {  ▷ new name matches p ◁
    if (r ≠ Λ) {  ▷ and also r: illegal ◁
      fputs(_("\n!␣Ambiguous␣prefix:␣matches␣<"), stdout); print_prefix_name(p);
      fputs(_(">\n␣and␣<"), stdout); print_prefix_name(r); err_print(">"); return name_dir;
        ▷ the unsection ◁
    }
    r ← p;  ▷ remember match ◁
    p ← p→llink;  ▷ try another ◁
    q ← r→rlink;  ▷ we'll get back here if the new p doesn't match ◁
  }
  if (p ≡ Λ) p ← q, q ← Λ;  ▷ q held the other branch of r ◁
}
```

This code is used in section 59.

**62:*** Although error messages are given in anomalous cases, we do return the unique best match when a discrepancy is found, because users often change a title in one place while forgetting to change it elsewhere.

```
⟨ If one match found, check for compatibility and return match 62* ⟩ ≡
  switch (section_name_cmp(&first, name_len, r)) {      ▷ compare all of r with new name ◁
  case prefix:
    if (¬ispref) {
      fputs(_("\n!␣New␣name␣is␣a␣prefix␣of␣<"), stdout); print_section_name(r); err_print(">");
    }
    else if (name_len < prefix_length(r)) set_prefix_length(r, name_len);
    /*␣fall␣through␣*/
  case equal: break;
  case extension:
    if (¬ispref ∨ first ≤ last) extend_section_name(r, first, last + 1, ispref);
    break;
  case bad_extension: fputs(_("\n!␣New␣name␣extends␣<"), stdout); print_section_name(r);
    err_print(">"); break;
  default:      ▷ no match: illegal ◁
    fputs(_("\n!␣Section␣name␣incompatible␣with␣<"), stdout); print_prefix_name(r);
    fputs(_(">,\n␣which␣abbreviates␣<"), stdout); print_section_name(r); err_print(">");
  }
  return r;
```

This code is used in section 59.

**67:*** The error locations can be indicated by using the global variables *loc*, *cur_line*, *cur_file_name* and *changing*, which tell respectively the first unlooked-at position in *buffer*, the current line number, the current file, and whether the current line is from *change_file* or *cur_file*. This routine should be modified on systems whose standard text editor has special line-numbering conventions.

```
⟨ Print error location based on input buffer 67* ⟩ ≡
  {
    char *k, *l;      ▷ pointers into buffer ◁
    if (changing ∧ include_depth ≡ change_depth)
      printf(_(".␣(l.␣%d␣of␣change␣file)\n"), change_line);
    else if (include_depth ≡ 0) printf(_(".␣(l.␣%d)\n"), cur_line);
    else printf(_(".␣(l.␣%d␣of␣include␣file␣%s)\n"), cur_line, cur_file_name);
    l ← (loc ≥ limit ? limit : loc);
    if (l > buffer) {
      for (k ← buffer; k < l; k++)
        if (*k ≡ '\t') putchar('␣');
        else putchar(*k);      ▷ print the characters already read ◁
      new_line;
      for (k ← buffer; k < l; k++) putchar('␣');      ▷ space out the next line ◁
    }
    for (k ← l; k < limit; k++) putchar(*k);      ▷ print the part not yet read ◁
    if (*limit ≡ '|') putchar('|');      ▷ end of C text in section names ◁
    putchar('␣');      ▷ to separate the message from future asterisks ◁
  }
```

This code is used in section 66.

**68.*** When no recovery from some error has been provided, we have to wrap up and quit as graciously as possible. This is done by calling the function *wrap_up* at the end of the code.

CTANGLE and CWEAVE have their own notions about how to print the job statistics. See the function(s) *print_stats* in the interface above and in the index.

On multi-tasking systems like the AMIGA it is very convenient to know a little bit more about the reasons why a program failed. The four levels of return indicated by the *history* value are very suitable for this purpose. Here, for instance, we pass the operating system a status of 0 if and only if the run was a complete success. Any warning or error message will result in a higher return value, so that AREXX scripts can be made sensitive to these conditions.

```
#define RETURN_OK 0      ▷ No problems, success ◁
#define RETURN_WARN 5    ▷ A warning only ◁
#define RETURN_ERROR 10  ▷ Something wrong ◁
#define RETURN_FAIL 20   ▷ Complete or severe failure ◁
  int wrap_up(void)
  {
    if (show_progress) new_line;
    if (show_stats) print_stats( );    ▷ print statistics about memory usage ◁
    ⟨ Print the job history 69* ⟩
    ⟨ Remove the temporary file if not already done 90* ⟩
    switch (history) {
    case spotless: return RETURN_OK;
    case harmless_message: return RETURN_WARN;
    case error_message: return RETURN_ERROR;
    case fatal_message: default: return RETURN_FAIL;
    }
  }
```

**69.*** ⟨ Print the job *history* 69* ⟩ ≡

```
  switch (history) {
  case spotless:
    if (show_happiness) puts(_("(No␣errors␣were␣found.)"));
    break;
  case harmless_message: puts(_("(Did␣you␣see␣the␣warning␣message␣above?)")); break;
  case error_message: puts(_("(Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong.)")); break;
  case fatal_message: default: puts(_("(That␣was␣a␣fatal␣error,␣my␣friend.)"));
  }
```

This code is used in section 68*.

**71.*** An overflow stop occurs if CWEB's tables aren't large enough.

```
  void overflow(const char *t)
  {
    printf(_("\n!␣Sorry,␣%s␣capacity␣exceeded"), t); fatal("", "");
  }
```

**73:* Command line arguments.** The user calls `CWEAVE` and `CTANGLE` with arguments on the command line. These are either file names or flags to be turned off (beginning with `"-"`) or flags to be turned on (beginning with `"+"`). TEX Live's `CWEB` executables accept several "long options" as well; see section ⟨ Handle flag argument 80* ⟩ for details. The following globals are for communicating the user's desires to the rest of the program. The various file name variables contain strings with the names of those files. Most of the 128 flags are undefined but available for future extensions.

```
⟨ Global variables 18* ⟩ +≡
  int argc;      ▷ copy of ac parameter to main ◁
  char **argv;      ▷ copy of av parameter to main ◁
  char C_file_name[max_file_name_length];      ▷ name of C_file ◁
  char tex_file_name[max_file_name_length];      ▷ name of tex_file ◁
  char idx_file_name[max_file_name_length];      ▷ name of idx_file ◁
  char scn_file_name[max_file_name_length];      ▷ name of scn_file ◁
  char check_file_name[max_file_name_length];      ▷ name of check_file ◁
  boolean flags[128];      ▷ an option for each 7-bit code ◁
```

**74:*** The *flags* will be initially *false*. Some of them are set to *true* before scanning the arguments; if additional flags are *true* by default they should be set before calling *common_init*.

```
⟨ Set the default options common to CTANGLE and CWEAVE 74* ⟩ ≡
  make_xrefs ← true;
```

This code is used in section 20*.

**75:*** We now must look at the command line arguments and set the file names accordingly. At least one file name must be present: the `CWEB` file. It may have an extension, or it may omit the extension to get `".w"` added. The TEX output file name is formed by replacing the `CWEB` file name extension by `".tex"`, and the C file name by replacing the extension by `".c"`, after removing the directory name (if any).

If there is a second file name present among the arguments, it is the change file, again either with an extension or without one to get `".ch"`. An omitted change file argument means that `"/dev/null"` or—on non-UNIX systems the contents of the compile-time variable `DEV_NULL` (TEX Live) or `_DEV_NULL` (Amiga)—should be used, when no changes are desired.

If there's a third file name, it will be the output file.

```
static void scan_args(void)
{
  char *dot_pos;  ▷ position of '.' in the argument ◁
  char *name_pos;  ▷ file name beginning, sans directory ◁
  register char *s;  ▷ register for scanning strings ◁
  boolean found_web ← false, found_change ← false, found_out ← false;
    ▷ have these names been seen? ◁
  strcpy(change_file_name, "/dev/null");
#if defined DEV_NULL
  strncpy(change_file_name, DEV_NULL, max_file_name_length − 2);
  change_file_name[max_file_name_length − 2] ← '\0';
#elif defined _DEV_NULL
  strncpy(change_file_name, _DEV_NULL, max_file_name_length − 2);
  change_file_name[max_file_name_length − 2] ← '\0';
#endif
  while (−−argc > 0) {
    if ((**(++argv) ≡ '-' ∨ **argv ≡ '+') ∧ *(*argv + 1)) ⟨ Handle flag argument 80* ⟩
    else {
      s ← name_pos ← *argv; dot_pos ← Λ;
      while (*s)
        if (*s ≡ '.') dot_pos ← s++;
        else if (*s ≡ DIR_SEPARATOR ∨ *s ≡ DEVICE_SEPARATOR ∨ *s ≡ '/')
          dot_pos ← Λ, name_pos ← ++s;
        else s++;
      if (¬found_web) ⟨ Make web_file_name, tex_file_name, and C_file_name 77* ⟩
      else if (¬found_change) ⟨ Make change_file_name 78 ⟩
      else if (¬found_out) ⟨ Override tex_file_name and C_file_name 79 ⟩
      else ⟨ Print usage error message and quit 81* ⟩
    }
  }
  if (¬found_web) ⟨ Print usage error message and quit 81* ⟩
}
```

**77*** We use all of $*argv$ for the *web_file_name* if there is a `'.'` in it, otherwise we add `".w"`. The other file names come from adding other things after the dot. We must check that there is enough room in *web_file_name* and the other arrays for the argument.

```
⟨ Make web_file_name, tex_file_name, and C_file_name 77* ⟩ ≡
  {
    if (s − *argv > max_file_name_length − 5) ⟨ Complain about argument length 82* ⟩
    if (dot_pos ≡ Λ) sprintf(web_file_name, "%s.w", *argv);
    else {
      strcpy(web_file_name, *argv); *dot_pos ← '\0';      ▷ string now ends where the dot was ◁
    }
    sprintf(tex_file_name, "%s.tex", name_pos);      ▷ strip off directory name ◁
    sprintf(idx_file_name, "%s.idx", name_pos); sprintf(scn_file_name, "%s.scn", name_pos);
    sprintf(C_file_name, "%s.c", name_pos); found_web ← true;
  }
```

This code is used in section 75*.

**80*** `#define` *flag_change* `(**argv ≠ '-')`

```
⟨ Handle flag argument 80* ⟩ ≡
  {
    if (strcmp("-help", *argv) ≡ 0 ∨ strcmp("--help", *argv) ≡ 0) ⟨ Display help message and exit 97* ⟩
    if (strcmp("-version", *argv) ≡ 0 ∨ strcmp("--version", *argv) ≡ 0)
      ⟨ Display version information and exit 100* ⟩
    if (strcmp("-verbose", *argv) ≡ 0 ∨ strcmp("--verbose", *argv) ≡ 0) strcpy(*argv, "-v");
    if (strcmp("-quiet", *argv) ≡ 0 ∨ strcmp("--quiet", *argv) ≡ 0) strcpy(*argv, "-q");
    for (dot_pos ← *argv + 1; *dot_pos > '\0'; dot_pos++) {
      switch (*dot_pos) {
      case 'v': show_banner ← show_progress ← show_happiness ← true; continue;
      case 'q': show_banner ← show_progress ← show_happiness ← false; continue;
      case 'd':
        if (sscanf(++dot_pos, "%u", &kpathsea_debug) ≠ 1) ⟨ Print usage error message and quit 81* ⟩
        while (isdigit(*dot_pos)) dot_pos++;      ▷ skip numeric part ◁
        dot_pos−−;      ▷ reset to final digit ◁
        continue;
      case 'l': use_language ← ++dot_pos; break;      ▷ from switch ◁
      default: flags[(eight_bits) *dot_pos] ← flag_change; continue;
      }
      break;      ▷ from for loop ◁
    }
  }
```

This code is cited in section 73*.

This code is used in section 75*.

**81*** ⟨ Print usage error message and quit 81* ⟩ ≡

```
cb_usage(program ≡ ctangle ? "ctangle" : program ≡ cweave ? "cweave" : "ctwill");
```

This code is used in sections 75* and 80*.

**82*** ⟨ Complain about argument length 82* ⟩ ≡

```
fatal(_("! Filename too long\n"), *argv);
```

This code is used in sections 77*, 78, and 79.

**83:* Output.** Here is the code that opens the output file:

```
⟨ Global variables 18* ⟩ +≡
  FILE *C_file;      ▷ where output of CTANGLE goes ◁
  FILE *tex_file;    ▷ where output of CWEAVE goes ◁
  FILE *idx_file;    ▷ where index from CWEAVE goes ◁
  FILE *scn_file;    ▷ where list of sections from CWEAVE goes ◁
  FILE *check_file;  ▷ temporary output file ◁
  FILE *active_file;  ▷ currently active file for CWEAVE output ◁
  char *found_filename;  ▷ filename found by kpse_find_file ◁
```

**84:*** ⟨ Scan arguments and open output files 84* ⟩ ≡

```
  scan_args();
  if (program ≡ ctangle) {
    if (check_for_change) ⟨ Open intermediate C output file 88* ⟩
    else if ((C_file ← fopen(C_file_name, "wb")) ≡ Λ)
      fatal(_("! Cannot open output file "), C_file_name);
  }
  else {
    if (check_for_change) ⟨ Open intermediate TEX output file 89* ⟩
    else if ((tex_file ← fopen(tex_file_name, "wb")) ≡ Λ)
      fatal(_("! Cannot open output file "), tex_file_name);
  }
```

This code is used in section 20*.

**85.* Extensions to CWEB.** The following sections introduce new or improved features that have been created by numerous contributors over the course of a quarter century.

Care has been taken to keep the original section numbering intact, so this new material should nicely integrate with the original "**85. Index**."

**86.* Language setting.** This global variable is set by the argument of the '`+l`' (or '`-l`') command-line option.

```
⟨ Global variables 18* ⟩ +≡
  const char *use_language ← "";      ▷ prefix of cwebmac.tex in TEX output ◁
```

**87* User communication.** The *scan_args* and *cb_show_banner* routines and the *bindtextdomain* argument string need a few extra variables.

```
#define max_banner 50
#define PATH_SEPARATOR separators[0]
#define DIR_SEPARATOR separators[1]
#define DEVICE_SEPARATOR separators[2]
⟨ Global variables 18* ⟩ +≡
  char cb_banner[max_banner];
  string texmf_locale;
#ifndef SEPARATORS
#define SEPARATORS "://"
#endif
  char separators[] ← SEPARATORS;
```

**88.* Temporary file output.** Most C projects are controlled by a `Makefile` that automatically takes care of the temporal dependecies between the different source modules. It may be convenient that `CWEB` doesn't create new output for all existing files, when there are only changes to some of them. Thus the `make` process will only recompile those modules where necessary. You can activate this feature with the '`+c`' command-line option. The idea and basic implementation of this mechanism can be found in the program `NUWEB` by Preston Briggs, to whom credit is due.

```
⟨ Open intermediate C output file 88* ⟩ ≡
  {
    if ((C_file ← fopen(C_file_name, "a")) ≡ Λ) fatal(_("! Cannot open output file "), C_file_name);
    else fclose(C_file);      ▷ Test accessability ◁
    strcpy(check_file_name, C_file_name);
    if (check_file_name[0] ≠ '\0') {
      char *dot_pos ← strrchr(check_file_name, '.');
      if (dot_pos ≡ Λ) strcat(check_file_name, ".ttp");
      else strcpy(dot_pos, ".ttp");
    }
    if ((C_file ← fopen(check_file_name, "wb")) ≡ Λ)
      fatal(_("! Cannot open output file "), check_file_name);
  }
```

This code is used in section 84*.

**89.*** ⟨ Open intermediate TEX output file 89* ⟩ ≡

```
  {
    if ((tex_file ← fopen(tex_file_name, "a")) ≡ Λ)
      fatal(_("! Cannot open output file "), tex_file_name);
    else fclose(tex_file);      ▷ Test accessability ◁
    strcpy(check_file_name, tex_file_name);
    if (check_file_name[0] ≠ '\0') {
      char *dot_pos ← strrchr(check_file_name, '.');
      if (dot_pos ≡ Λ) strcat(check_file_name, ".wtp");
      else strcpy(dot_pos, ".wtp");
    }
    if ((tex_file ← fopen(check_file_name, "wb")) ≡ Λ)
      fatal(_("! Cannot open output file "), check_file_name);
  }
```

This code is used in section 84*.

**90.*** Before we leave the program we have to make sure that the output files are correctly written.

```
⟨ Remove the temporary file if not already done 90* ⟩ ≡
  if (C_file) fclose(C_file);
  if (tex_file) fclose(tex_file);
  if (check_file) fclose(check_file);
  if (strlen(check_file_name))      ▷ Delete the temporary file in case of a break ◁
    remove(check_file_name);
```

This code is used in section 68*.

**91* Internationalization.** If translation catalogs for your personal `LANGUAGE` are installed at the appropriate place, `CTANGLE` and `CWEAVE` will talk to you in your favorite language. Catalog `cweb` contains all strings from "plain `CWEB`," catalog `cweb-tl` contains a few extra strings specific to the TEX Live interface, and catalog `web2c-help` contains the "`--help`" texts for `CTANGLE` and `CWEAVE`.

If such translation files are not available, you may want to improve this system by checking out the sources and translating the strings in files `cweb.pot`, `cweb-tl.pot`, and `web2c-help.pot`, and submitting the resulting `*.po` files to the maintainers at `tex-k@tug.org`.

**Note to maintainers:** `CWEB` in TEX Live generally does *not* set `HAVE_GETTEXT` at build-time, so `i18n` is "off" by default. If you want to create `CWEB` executables with NLS support, you have to recompile the TEX Live sources with a positive value for `HAVE_GETTEXT` in `comm-w2c.h`. Also you have to "compile" the NLS catalogs provided for `CWEB` in the source tree with `msgfmt` and store the resulting `.mo` files at an appropriate place in the file system.

Plans for TEX Live are to store NLS catalogs inside the "TEX Directory Structure" (TDS) and look them up with the help of the configuration variable "`TEXMFLOCALEDIR`," which should contain a single absolute path definition. Below we use the KPATHSEA function *kpse_var_expand* to evaluate this variable from various origins and redirect the "GNU gettext utilities" to a possibly different location than the canonical `/usr/share/locale`.

There are several ways to set `TEXMFLOCALEDIR`:

(a) a user-set environment variable `TEXMFLOCALEDIR` (overridden by `TEXMFLOCALEDIR_cweb`);
(b) a line in KPATHSEA configuration file `texmf.cnf`, e.g., `TEXMFLOCALEDIR=$TEXMFMAIN/locale` or `TEXMFLOCALEDIR.cweb=$TEXMFMAIN/locale`.

```
⟨ Include files 3* ⟩ +≡
#if HAVE_GETTEXT
#include <locale.h>     ▷ LC_MESSAGES, LC_CTYPE ◁
#else
#define setlocale(a, b) ""
#define bindtextdomain(a, b) ""
#define textdomain(a) ""
#endif
```

**92*** ⟨ Set locale and bind language catalogs 92* ⟩ ≡

```
setlocale(LC_MESSAGES, setlocale(LC_CTYPE, ""));
texmf_locale ← kpse_var_expand("${TEXMFLOCALEDIR}");
bindtextdomain("cweb", bindtextdomain("cweb-tl", bindtextdomain("web2c-help",
    strcmp(texmf_locale, "") ? texmf_locale : "/usr/share/locale")));  free(texmf_locale);
textdomain("cweb");     ▷ the majority of "strings" come from "plain CWEB" ◁
```

This code is used in section 20*.

**93* File lookup with** KPATHSEA**.** The `CTANGLE` and `CWEAVE` programs from the original `CWEB` package use the compile-time default directory or the value of the environment variable `CWEBINPUTS` as an alternative place to be searched for files, if they could not be found in the current directory.

This version uses the KPATHSEA mechanism for searching files. The directories to be searched for come from three sources:

(a) a user-set environment variable `CWEBINPUTS` (overridden by `CWEBINPUTS_cweb`);
(b) a line in KPATHSEA configuration file `texmf.cnf`,
e.g., `CWEBINPUTS=$TEXMFDOTDIR:$TEXMF/texmf/cweb//`
or `CWEBINPUTS.cweb=$TEXMFDOTDIR:$TEXMF/texmf/cweb//`;
(c) compile-time default directories (specified in `texmf.in`),
i.e., `$TEXMFDOTDIR:$TEXMF/texmf/cweb//`.

**#define** *kpse_find_cweb*(*name*) *kpse_find_file*(*name*, *kpse_cweb_format*, *true*)

⟨ Include files 3* ⟩ +≡
**#include** `<kpathsea/kpathsea.h>`
▷ include every KPATHSEA header; *kpathsea_debug*, **const_string**, **string** ◁
**#include** `<w2c/config.h>` ▷ **integer** ◁
**#include** `<lib/lib.h>` ▷ *versionstring* ◁

**94*** We set *kpse_program_name* to '`cweb`'. This means if the variable `CWEBINPUTS.cweb` is present in `texmf.cnf` (or `CWEBINPUTS_cweb` in the environment) its value will be used as the search path for filenames. This allows different flavors of `CWEB` to have different search paths.

⟨ Set up `PROGNAME` feature and initialize the search path mechanism 94* ⟩ ≡
*kpse_set_program_name*(*argv*[0], `"cweb"`);
This code is used in section 20*.

**95*** When the files you expect are not found, the thing to do is to enable KPATHSEA runtime debugging by assigning to the *kpathsea_debug* variable a small number via the '`-d`' option. The meaning of this number is shown below. To set more than one debugging option, simply sum the corresponding numbers.

| | |
|---|---|
| 1 | report '`stat`' calls |
| 2 | report lookups in all hash tables |
| 4 | report file openings and closings |
| 8 | report path information |
| 16 | report directory list |
| 32 | report on each file search |
| 64 | report values of variables being looked up |

Debugging output is always written to *stderr*, and begins with the string '`kdebug:`'.

**96.* System dependent changes.** The most volatile stuff comes at the very end.
Modules for dealing with help messages and version info.

```
⟨ Include files 3* ⟩ +≡
#define CWEB
#include "help.h"    ▷ CTANGLEHELP, CWEAVEHELP, CTWILLHELP ◁
```

**97.*** ⟨ Display help message and *exit* 97* ⟩ ≡

```
cb_usagehelp(program ≡ ctangle ? CTANGLEHELP : program ≡ cweave ? CWEAVEHELP : CTWILLHELP);
```

This code is used in section 80*.

**98.*** Special variants from Web2c's '`lib/usage.c`', adapted for `i18n/t10n`. We simply filter the strings through the catalogs (if available).

```
⟨ Predeclaration of procedures 7* ⟩ +≡
  static void cb_usage(const_string str);
  static void cb_usagehelp(const_string *message);
```

**99.***

```
  static void cb_usage(const_string str)
  {
    textdomain("cweb-tl"); fprintf(stderr, _("%s: Need one to three file arguments.\n"), str);
    fprintf(stderr, _("Try `%s --help' for more information.\n"), str); textdomain("cweb");
    history ← fatal_message; exit(wrap_up());
  }
  static void cb_usagehelp(const_string *message)
  {
    textdomain("web2c-help");
    while (*message) {    ▷ empty string "" has special meaning for gettext ◁
      printf("%s\n", strcmp("", *message) ? _(*message) : *message); ++message;
    }
    textdomain("cweb-tl");
    printf(_("\nPackage home page: %s.\n"), "https://ctan.org/pkg/cweb"); textdomain("cweb");
    history ← spotless; exit(wrap_up());
  }
```

**100.*** The version information will not be translated, it uses a generic text template in English.

```
⟨ Display version information and exit 100* ⟩ ≡
  printversionandexit(cb_banner,
      program ≡ ctwill ? "Donald E. Knuth" : "Silvio Levy and Donald E. Knuth", Λ,
      "Contemporary development on https://github.com/ascherer/cweb.\n");
```

This code is used in section 80*.

**101.*** But the "banner" is, at least the first part.

```
  void cb_show_banner(void)
  {
    textdomain("cweb-tl"); printf("%s%s\n", _(cb_banner), versionstring); textdomain("cweb");
  }
```

## 102* Index.

The following sections were changed by the change file: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20, 23, 25, 29, 30, 32, 34, 36, 38, 39, 41, 42, 46, 51, 57, 58, 60, 62, 67, 68, 69, 71, 73, 74, 75, 77, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102.

⟨ Common code for `CWEAVE` and `CTANGLE` 2*, 4*, 5*, 6*, 8*, 9*, 11*, 13*, 14* ⟩ Used in section 1*.
⟨ Complain about argument length 82* ⟩ Used in sections 77*, 78, and 79.
⟨ Compute the hash code $h$ 49 ⟩ Used in section 48.
⟨ Compute the name location $p$ 50 ⟩ Used in section 48.
⟨ Display help message and *exit* 97* ⟩ Used in section 80*.
⟨ Display version information and *exit* 100* ⟩ Used in section 80*.
⟨ Enter a new name into the table at position $p$ 51* ⟩ Used in section 48.
⟨ Global variables 18*, 19, 21, 22, 25*, 26, 37, 43, 44, 46*, 65, 73*, 83*, 86*, 87* ⟩ Used in section 1*.
⟨ Handle flag argument 80* ⟩ Cited in section 73*. Used in section 75*.
⟨ If no match found, add new name to tree 61 ⟩ Used in section 59.
⟨ If one match found, check for compatibility and return match 62* ⟩ Used in section 59.
⟨ If the current line starts with `@y`, report any discrepancies and **return** 34* ⟩ Used in section 32*.
⟨ Include files 3*, 91*, 93*, 96* ⟩ Used in section 1*.
⟨ Initialize pointers 45, 47 ⟩ Used in section 20*.
⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 60* ⟩ Used in section 59.
⟨ Make *change_file_name* 78 ⟩ Used in section 75*.
⟨ Make *web_file_name*, *tex_file_name*, and *C_file_name* 77* ⟩ Used in section 75*.
⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 31 ⟩ Used in sections 27 and 32*.
⟨ Open input files 36* ⟩ Used in section 35.
⟨ Open intermediate C output file 88* ⟩ Used in section 84*.
⟨ Open intermediate TEX output file 89* ⟩ Used in section 84*.
⟨ Override *tex_file_name* and *C_file_name* 79 ⟩ Used in section 75*.
⟨ Predeclaration of procedures 7*, 10*, 12*, 15*, 24, 28, 33, 55, 64, 76, 98* ⟩ Used in section 1*.
⟨ Print error location based on input buffer 67* ⟩ Used in section 66.
⟨ Print the job *history* 69* ⟩ Used in section 68*.
⟨ Print usage error message and quit 81* ⟩ Used in sections 75* and 80*.
⟨ Read from *change_file* and maybe turn off *changing* 41* ⟩ Used in section 38*.
⟨ Read from *cur_file* and maybe turn on *changing* 40 ⟩ Used in section 38*.
⟨ Remove the temporary file if not already done 90* ⟩ Used in section 68*.
⟨ Scan arguments and open output files 84* ⟩ Used in section 20*.
⟨ Set locale and bind language catalogs 92* ⟩ Used in section 20*.
⟨ Set the default options common to `CTANGLE` and `CWEAVE` 74* ⟩ Used in section 20*.
⟨ Set up `PROGNAME` feature and initialize the search path mechanism 94* ⟩ Used in section 20*.
⟨ Skip over comment lines in the change file; **return** if end of file 29* ⟩ Used in section 27.
⟨ Skip to the next nonblank line; **return** if end of file 30* ⟩ Used in section 27.
⟨ Try to open include file, abort push if unsuccessful, go to *restart* 39* ⟩ Used in section 38*.