# The TIE processor

(CWEB Version 2.4 [TEX Live])






March 14, 2024 at 19:02

**1* Introduction.**
Whenever a programmer wants to change a given `WEB` or `CWEB` program (referred to as a `WEB` program throughout this program) because of system dependencies, she or he will create a new change file. In addition there may be a second change file to modify system independent modules of the program. But the `WEB` file cannot be tangled and weaved with more than one change file simultaneously. Therefore, we introduce the present program to merge a `WEB` file and several change files producing a new `WEB` file. Since the input files are tied together, the program is called `TIE`. Furthermore, the program can be used to merge several change files giving a new single change file. This method seems to be more important because it doesn't modify the original source file. The use of `TIE` can be expanded to other programming languages since this processor only knows about the structure of change files and does not interpret the master file at all.

The program `TIE` has to read lines from several input files to bring them in some special ordering. For this purpose an algorithm is used which looks a little bit complicated. But the method used only needs one buffer line for each input file. Thus the storage requirement of `TIE` does not depend on the input data.

The program is written in C and uses only few features of a particular environment that may need to be changed in other installations. The changes needed may refer to the access of the command line if this can be not supported by any C compiler.

The "banner line" defined here should be changed whenever `TIE` is modified. This program is put into the public domain. Nevertheless the copyright notice must not be replaced or modified.

```
#define banner "This␣is␣TIE,␣CWEB␣Version␣2.4"
        ▷ will be extended by the TEX Live versionstring ◁
#define copyright "Copyright␣(c)␣1989,1992␣by␣THD/ITI.␣All␣rights␣reserved."
```

**2*** The main outline of the program is given in the next section. This can be used more or less for any C program.

```
⟨ Global #includes 15* ⟩
⟨ Global constants 5* ⟩
⟨ Global types 7 ⟩
⟨ Global variables 6* ⟩
⟨ Error handling functions 31* ⟩
⟨ Internal functions 24* ⟩
⟨ The main function 59* ⟩
```

**3*** Here are some macros for common programming idioms.

```
#define loop while (1)      ▷ repeat over and over until a break happens ◁
#define do_nothing          ▷ empty statement ◁
  format loop while
```

**4*** The types **boolean** (with values *false* and *true*) and **string** come from `<kpathsea/simpletypes.h>`.

**5*** The following parameters should be sufficient for most applications of `TIE`.

```
⟨ Global constants 5* ⟩ ≡
#define buf_size 512        ▷ maximum length of one input line ◁
#define max_file_index 32
    ▷ we don't think that anyone needs more than 32 change files, but ... just change it ◁
```

This code is used in section 2*.

**6*** We introduce a history variable that allows us to set a return code if the operating system can use it. First we introduce the coded values for the history. This variable must be initialized. (We do this even if the value given may be the default for variables, just to document the need for the initial value.)

```
⟨ Global variables 6* ⟩ ≡
  typedef enum {
    spotless, troublesome, fatal
  } return_code;
  static return_code history ← spotless;
```

See also sections 9, 22, 23, 26, and 35.

This code is used in section 2*.

**15* Input and output.**
Output for the user is done by writing on file *term_out*, which is assumed to consist of characters of type **text_char**. It should be linked to *stdout* usually. Terminal input is not needed in this version of `TIE`. *stdin* and *stdout* are predefined if we include the `stdio.h` definitions. Although I/O redirection for *stdout* is usually available you may lead output to another file if you change the definition of *term_out*. Also we define some macros for terminating an output line and writing strings to the user.

```
#define term_out stdout
#define print(a) fprintf(term_out,"%s",a)   ▷ 'print' means write on the terminal ◁
#define print2(a,b) fprintf(term_out,a,b)   ▷ same with two arguments ◁
#define print3(a,b,c) fprintf(stderr,a,b,c)   ▷ same with three arguments ◁
#define print_c(v) fputc(v,term_out);   ▷ print a single character ◁
#define new_line(v) fputc('\n',v)   ▷ start new line ◁
#define term_new_line new_line(term_out)   ▷ start new line of the terminal ◁
#define print_ln(v)
        {
          fprintf(term_out,"%s",v); term_new_line;
        }   ▷ 'print' and then start new line ◁
#define print2_ln(a,b)
        {
          print2(a,b); term_new_line;
        }   ▷ same with two arguments ◁
#define print3_ln(a,b,c)
        {
          print3(a,b,c); new_line(stderr);
        }   ▷ same with three arguments ◁
#define print_nl(v)
        {
          term_new_line; print(v);
        }   ▷ print information starting on a new line ◁
#define print2_nl(a,b)
        {
          term_new_line; print2(a,b);
        }   ▷ same for two arguments ◁
```

⟨ Global **#include**s 15* ⟩ ≡

```
#include "cpascal.h"   ▷ decr and incr ◁
#include <kpathsea/kpathsea.h>
#define usage tieusage   ▷ Also redefine usage to avoid clash with function from lib. ◁
```

This code is used in section 2*.

**16*** And we need dynamic memory allocation. This should cause no trouble in any C program. The `kpathsea` include files handle the definition of *malloc*, too.

**18.* Data structures.**
The multiple input files (master file and change files) are treated the same way. To organize the simultaneous usage of several input files, we introduce the data type **in_file_modes**.

The mode *search* indicates that `TIE` searches for a match of the input line with any line of an input file in *reading* mode. *test* is used whenever a match is found and it has to be tested if the next input lines do match also. *reading* describes that the lines can be read without any check for matching other lines. *ignore* denotes that the file cannot be used. This may happen because an error has been detected or because the end of the file has been found.

**file_types** is used to describe whether a file is a master file or a change file. The value *unknown* is added to this type to set an initial mode for the output file. This enables us to check whether any option was used to select the kind of output. (This would even be necessary if we would assume a default action for missing options.)

```
⟨ Global types 7 ⟩ +≡
  typedef enum {
    search, test, reading, ignore
  } in_file_modes;
  typedef enum {
    unknown, master, chf
  } file_types;
```

**19.*** A variable of type **out_md_type** will tell us in what state the output change file is during processing. *normal* will be the state, when we did not yet start a change, *pre* will be set when we write the lines to be changes and *post* will indicate that the replacement lines are written.

```
⟨ Global types 7 ⟩ +≡
  typedef enum {
    normal, pre, post
  } out_md_type;
```

**24.* File I/O.**
The basic function *get_line* can be used to get a line from an input file. The line is stored in the *buffer* part of the descriptor. The components *limit* and *line* are updated. If the end of the file is reached *mode* is set to *ignore*. On some systems it might be useful to replace tab characters by a proper number of spaces since several editors used to create change files insert tab characters into a source file not under control of the user. So it might be a problem to create a matching change file.

We define *get_line* to read a line from a file specified by the corresponding file descriptor.

```
⟨ Internal functions 24* ⟩ ≡
  static void get_line(file_index i)
  {
    register input_description *inp_desc ← input_organization[i];
    if (inp_desc→mode ≡ ignore) return;
    if (feof(inp_desc→the_file)) ⟨ Handle end of file and return 25 ⟩
    ⟨ Get line into buffer 27* ⟩
  }
```

See also sections 38*, 39*, 42*, 43*, 44*, and 55*.

This code is used in section 2*.

**27.*** Lines must fit into the buffer completely. We read all characters sequentially until an end of line is found (but do not forget to check for `EOF`!). Too long input lines will be truncated. This will result in a damaged output if they occur in the replacement part of a change file, or in an incomplete check if the matching part is concerned. Tab character expansion might be done here.

```
⟨ Get line into buffer 27* ⟩ ≡
  {
    int final_limit;    ▷ used to delete trailing spaces ◁
    int c;    ▷ the actual character read ◁
    ⟨ Increment the line number and print a progess report at certain times 28 ⟩
    inp_desc→limit ← final_limit ← 0;
    while (inp_desc→limit < buf_size) {
      c ← fgetc(inp_desc→the_file);
      ⟨ Check c for EOF, return if line was empty, otherwise break to process last line 29* ⟩
      inp_desc→buffer[inp_desc→limit++] ← c ← map_xord(c);
      if (c ≡ nl_mark) break;    ▷ end of line found ◁
      if (c ≠ '␣' ∧ c ≠ tab_mark ∧ c ≠ '\r') final_limit ← inp_desc→limit;
    }
    ⟨ Test for truncated line, skip to end of line 30 ⟩
    inp_desc→limit ← final_limit;
  }
```

This code is used in section 24*.

**29:*** There may be incomplete lines if the editor used does not make sure that the last character before end of file is an end of line. In such a case we must process the final line. If the current line is empty, we just can **return**. Note that this test must be done *before* the character read is translated.

```
⟨ Check c for EOF, return if line was empty, otherwise break to process last line 29* ⟩ ≡
  if (c ≡ EOF) {
    if (inp_desc→limit ≤ 0) {
      inp_desc→mode ← ignore; inp_desc→limit ← −1;      ▷ mark end-of-file ◁
      if (inp_desc→type_of_file ≡ master) input_has_ended ← true;
      return;
    }
    else {      ▷ add end of line mark ◁
      c ← nl_mark; break;
    }
  }
```

This code is used in section 27*.

**31.* Reporting errors to the user.**
There may be errors if a line in a given change file does not match a line in the master file or a replacement in a previous change file. Such errors are reported to the user by saying

$$err\_print(\texttt{"!␣Error␣message"})(file\_no);$$

where *file_no* is the number of the file which is concerned by the error. Please note that no trailing dot is supplied by the error message because it is appended by *err_print*.

This function is implemented as a macro. It gives a message and an indication of the offending file. The actions to determine the error location are provided by a function called *err_loc*.

```
#define error_loc(m)  err_loc(m); history ← troublesome; }
#define err_print(m)  { new_line(stderr); fprintf(stderr, "%s", m); error_loc
```

⟨ Error handling functions 31* ⟩ ≡

```
static void err_loc(int i)      ▷ prints location of error ◁
{
  print3_ln("␣(file␣%s,␣l.%ld).", input_organization[i]¬name_of_file, input_organization[i]¬line);
}
```

This code is used in section 2*.

**32.*** Non recoverable errors are handled by calling *fatal_error* that outputs a message and then calls '*jump_out*'. *err_print* will print the error message followed by an indication of where the error was spotted in the source files. *fatal_error* cannot state any files because the problem is usually to access these.

```
#define fatal_error(m)
        {
          fprintf(stderr, "%s", m); fputc('.', stderr); history ← fatal; new_line(stderr); jump_out();
        }
```

**33.*** *jump_out* just cuts across all active procedure levels and jumps out of the program. It is used when no recovery from a particular error has been provided. The return code from this program should be regarded by the caller.

```
#define jump_out()  exit(EXIT_FAILURE)
```

**34.* Handling multiple change files.**
In the standard version we take the name of the files from the command line. It is assumed that filenames can be used as given in the command line without changes.

First there are some sections to open all files. If a file is not accessible, the run will be aborted. Otherwise the name of the open file will be displayed.

```
⟨ Prepare the output file 34* ⟩ ≡
  {
    out_file ← fopen(out_name, "wb");
    if (out_file ≡ Λ) {
      fatal_error("! Could not open/create output file");
    }
  }
```

This code is used in section 59*.

**36.*** For the master file we start just reading its first line into the buffer, if we could open it.

```
⟨ Get the master file started 36* ⟩ ≡
  {
    input_organization[0]→the_file ← kpse_open_file(input_organization[0]→name_of_file, kpse_web_format);
    if (input_organization[0]→the_file ≡ Λ) fatal_error("! Could not open master file");
    print2("(%s)", input_organization[0]→name_of_file); term_new_line;
    input_organization[0]→type_of_file ← master; get_line(0);
  }
```

This code is used in section 59*.

**37.*** For the change files we must skip the comment part and see, whether we can find any change in it. This is done by *init_change_file*.

```
⟨ Prepare the change files 37* ⟩ ≡
  {
    file_index i;
    i ← 1;
    while (i < no_ch) {
      input_organization[i]→the_file ← kpse_open_file(input_organization[i]→name_of_file, kpse_web_format);
      if (input_organization[i]→the_file ≡ Λ) fatal_error("! Could not open change file");
      print2("(%s)", input_organization[i]→name_of_file); term_new_line; init_change_file(i, true); incr(i);
    }
  }
```

This code is used in section 59*.

**38.* Input/output organization.**
Here's a simple function that checks if two lines are different.

```
⟨ Internal functions 24* ⟩ +≡
  static boolean lines_dont_match(file_index i, file_index j)
  {
    buffer_index k, lmt;
    if (input_organization[i]¬limit ≠ input_organization[j]¬limit) return (true);
    lmt ← input_organization[i]¬limit;
    for (k ← 0; k < lmt; k++)
      if (input_organization[i]¬buffer[k] ≠ input_organization[j]¬buffer[k]) return (true);
    return (false);
  }
```

**39.*** Function *init_change_file*$(i, b)$ is used to ignore all lines of the input file with index $i$ until the next change module is found. The boolean parameter $b$ indicates whether we do not want to see `@x` or `@y` entries during our skip.

```
⟨ Internal functions 24* ⟩ +≡
  static void init_change_file(file_index i, boolean b)
  {
    register input_description *inp_desc ← input_organization[i];
    ⟨ Skip over comment lines; return if end of file 40 ⟩
    ⟨ Skip to the next nonblank line; return if end of file 41 ⟩
  }
```

**42.*** The *put_line* function is used to write a line from input buffer $j$ to the output file.

```
⟨ Internal functions 24* ⟩ +≡
  static void put_line(file_index j)
  {
    buffer_index i;      ▷ index into the buffer ◁
    buffer_index lmt;    ▷ line length ◁
    ASCII_Code *p;       ▷ output pointer ◁
    lmt ← input_organization[j]¬limit; p ← input_organization[j]¬buffer;
    for (i ← 0; i < lmt; i++) fputc(map_xchr(*p++), out_file);
    new_line(out_file);
  }
```

**43.*** The function *e_of_ch_module* returns true if the input line from file $i$ starts with `@z`.

```
⟨ Internal functions 24* ⟩ +≡
  static boolean e_of_ch_module(file_index i)
  {
    register input_description *inp_desc ← input_organization[i];
    if (inp_desc¬limit < 0) {
      print_nl("! At the end of change file missing @z ");
      print2("%s", input_organization[i]¬name_of_file); term_new_line; return (true);
    }
    else if (inp_desc¬limit ≥ 2)
      if (inp_desc¬buffer[0] ≡ '@' ∧ (inp_desc¬buffer[1] ≡ 'Z' ∨ inp_desc¬buffer[1] ≡ 'z'))
        return (true);
    return (false);
  }
```

**44:*** The function *e_of_ch_preamble* returns *true* if the input line from file $i$ starts with `@y`.

```
⟨ Internal functions 24* ⟩ +≡
  static boolean e_of_ch_preamble(file_index i)
  {
    register input_description *inp_desc ← input_organization[i];
    if (inp_desc→limit ≥ 2 ∧ inp_desc→buffer[0] ≡ '@')
      if (inp_desc→buffer[1] ≡ 'Y' ∨ inp_desc→buffer[1] ≡ 'y') return (true);
    return (false);
  }
```

**47:*** Now we will set *test_input* to the file that has another match for the current line. This depends on the state of the other change files. If no other file matches, *actual_input* refers to a line to write and *test_input* is set to *none*.

```
#define none (max_file_index + 1)
⟨ Scan all other files for changes to be done 47* ⟩ ≡
  test_input ← none; test_file ← actual_input;
  while (test_input ≡ none ∧ test_file < no_ch − 1) {
    incr(test_file);
    switch (input_organization[test_file]→mode) {
    case search:
      if (lines_dont_match(actual_input, test_file) ≡ false) {
        input_organization[test_file]→mode ← test; test_input ← test_file;
      }
      break;
    case test:
      if (lines_dont_match(actual_input, test_file) ≡ true) {      ▷ error, sections do not match ◁
        input_organization[test_file]→mode ← search;
        err_print("! Sections do not match")(actual_input); err_loc(test_file);
        init_change_file(test_file, false);
      }
      else test_input ← test_file;
      break;
    case reading: do_nothing;      ▷ this can't happen ◁
      break;
    case ignore: do_nothing;      ▷ nothing to do ◁
      break;
    }
  }
```

This code is used in section 45.

**48:*** For the output we must distinguish whether we create a new change file or a new master file. The change file creation needs some closer inspection because we may be before a change, in the pattern part or in the replacement part. For a master file we have to write the line from the current actual input.

```
⟨ Handle output 48* ⟩ ≡
  if (prod_chf ≡ chf) loop {
    ⟨ Test for normal, break when done 49 ⟩
    ⟨ Test for pre, break when done 50 ⟩
    ⟨ Test for post, break when done 51 ⟩
  }
  else
    if (test_input ≡ none) put_line(actual_input);
```

This code is used in section 45.

**53:*** To create the new output file we have to scan the whole master file and all changes in effect when it ends. At the very end it is wise to check for all changes to have completed–in case the last line of the master file was to be changed.

```
⟨ Process the input 53* ⟩ ≡
  actual_input ← 0; input_has_ended ← false;
  while (input_has_ended ≡ false ∨ actual_input ≠ 0)
    ⟨ Process a line, break when end of source reached 45 ⟩
  if (out_mode ≡ pre) {      ▷ last line has been deleted ◁
    fputc(map_xchr('@'@'), out_file); fputc(map_xchr('@'y'), out_file); new_line(out_file);
    out_mode ← post;
  }
  if (out_mode ≡ post) {      ▷ last line has been changed ◁
    fputc(map_xchr('@'@'), out_file); fputc(map_xchr('@'z'), out_file); new_line(out_file);
  }
```

This code is used in section 59*.

**55:*** We want to tell the user about our command line options. This is done by the *usage*( ) function. It contains merely the necessary print statement and exits afterwards.

```
⟨ Internal functions 24* ⟩ +≡
  static void usage(void)
  {
    print("Usage: tie -m|-c outfile master changefile(s)"); term_new_line; jump_out( );
  }
```

**56.*** We must scan through the list of parameters, given in *argv*. The number is in *argc*. We must pay attention to the flag parameter. We need at least 5 parameters and can handle up to *max_file_index* change files. The names of the file parameters will be inserted into the structure of *input_organization*. The first file is special. It indicates the output file. When we allow flags at any position, we must find out which name is for what purpose. The master file is already part of the *input_organization* structure (index 0). As long as the number of files found (counted in *no_ch*) is $-1$ we have not yet found the output file name.

```
⟨ Scan the parameters 56* ⟩ ≡
  {
    int act_arg;
    if (argc < 5 ∨ argc > max_file_index + 4 − 1) usage( );
    no_ch ← −1;      ▷ fill this part of input_organization ◁
    for (act_arg ← 1; act_arg < argc; act_arg++) {
      if (argv[act_arg][0] ≡ '-') ⟨ Set a flag 57 ⟩
      else ⟨ Get a file name 58 ⟩
    }
    if (no_ch ≤ 0 ∨ prod_chf ≡ unknown) usage( );
  }
```

This code is used in section 59*.

**59* The main program.**
Here is where TIE starts, and where it ends.

This version of the TIE program uses the KPATHSEA library for searching files. Firstly, we use the *kpse_web_format* when opening input files, which triggers the inspection of the WEBINPUTS environment variable. Secondly, we set *kpse_program_name* to 'tie'. This means if the variable WEBINPUTS.tie is present in texmf.cnf (or WEBINPUTS_tie in the environment) its value will be used as the search path for filenames. This allows different flavors of TIE (or other WEB programs) to have different search paths. In all, the directories to be searched for come from at least two sources:

(a) a user-set environment variable WEBINPUTS (overridden by WEBINPUTS_tie);
(b) a line in KPATHSEA configuration file texmf.cnf,
e.g., WEBINPUTS=$TEXMFDOTDIR:$TEXMF/texmf/web//
or WEBINPUTS.tie=$TEXMFDOTDIR:$TEXMF/texmf/web//.

Note that, although WEBINPUTS might suggest otherwise, TIE is more or less language-agnostic and that it is perfectly capable of handling CWEB files as input as well, as long as the "change files" adhere to the general @x, @y, @z convention.

```
⟨ The main function 59* ⟩ ≡
  int main(int argc, string *argv)
  {
    {
      ⟨ Local variables for initialisation 12 ⟩
      ⟨ Set initial values 10 ⟩
    }
    kpse_set_program_name(argv[0], "tie"); print(banner);     ▷ print a "banner line" ◁
    print_ln(versionstring);     ▷ Web2C version ◁
    print_ln(copyright);     ▷ include the copyright notice ◁
    actual_input ← 0; out_mode ← normal; ⟨ Scan the parameters 56* ⟩
    ⟨ Prepare the output file 34* ⟩
    ⟨ Get the master file started 36* ⟩
    ⟨ Prepare the change files 37* ⟩
    ⟨ Process the input 53* ⟩
    ⟨ Check that all changes have been read 54 ⟩
    ⟨ Print the job history 60* ⟩
  }
```

This code is used in section 2*.

**60*** We want to pass the *history* value to the operating system so that it can be used to govern whether or not other programs are started. Additionally we report the history to the user, although this may not be "UNIX" style—but we are in best companion: WEB and TEX do the same.

```
⟨ Print the job history 60* ⟩ ≡
  {
    switch (history) {
    case spotless: print2_nl("(%s.)", "No␣errors␣were␣found"); term_new_line; break;
    case troublesome: new_line(stderr);
      fprintf(stderr, "(Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong.)"); new_line(stderr);
      break;
    case fatal: default:     ▷ Anything except spotless, troublesome, or fatal is a bug. ◁
      new_line(stderr); fprintf(stderr, "(That␣was␣a␣fatal␣error,␣my␣friend.)"); new_line(stderr);
      break;
    }
    exit(history ≡ spotless ? EXIT_SUCCESS : EXIT_FAILURE);
  }
```

This code is used in section 59*.

**62.* Index.**
Here is the cross-reference table for the TIE processor.

The following sections were changed by the change file: 1, 2, 3, 4, 5, 6, 15, 16, 18, 19, 24, 27, 29, 31, 32, 33, 34, 36, 37, 38, 39, 42, 43, 44, 47, 48, 53, 55, 56, 59, 60, 62.