The TIE processor

(CWEB Version 2.4 [TeX Live])

Introduction ................................................................. 1 1
The character set .......................................................... 7 2
Input and output ........................................................... 15 3
Data structures .............................................................. 18 4
File I/O ................................................................. 24 5
Reporting errors to the user .............................................. 31 6
Handling multiple change files ....................................... 34 7
Input/output organization ............................................... 38 8
The main program ........................................................ 59 11
System-dependent changes ............................................. 61 11
Index ............................................................................. 62 12

Copyright © 1989, 1992 by Technische Hochschule Darmstadt,
Fachbereich Informatik, Institut für Theoretische Informatik
All rights reserved.

This program is distributed WITHOUT ANY WARRANTY, express or implied.
Permission is granted to make and distribute verbatim copies of this program provided that the copyright
notice and this permission notice are preserved on all copies.
Permission is granted to copy and distribute modified versions of this program under the conditions for ver-
batim copying, provided that the entire resulting derived work is distributed under the terms of a permission
notice identical to this one.

March 17, 2021 at 13:08
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 not be 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.

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

3* Here are some macros for common programming idioms.

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

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

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§) ≡

```c
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.

```c
#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 #includes 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.)

```c
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.

```c
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.

\[
\text{get_line(file_index i)}
\]

\[
\text{static void get_line(file_index i)}
\]

\[
\text{\{}
\]

\[
\text{\quad register input_description *inp_desc \leftarrow input_organisation[i]};
\]

\[
\text{\quad if (inp_desc-mode \equiv ignore) return;}
\]

\[
\text{\quad if (feof(inp_desc-the_file)) \langle \text{Handle end of file and return 25} \rangle}
\]

\[
\text{\quad \langle \text{Get line into buffer 27*} \rangle}
\]

\[
\text{\}}
\]

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.

\[
\text{\{ \text{Get line into buffer 27*} \}}
\]

\[
\text{\{}
\]

\[
\text{\quad int final_limit; \quad \text{\textit{used to delete trailing spaces \triangleleft}}}
\]

\[
\text{\quad int c; \quad \text{\textit{the actual character read \triangleleft}}}
\]

\[
\text{\quad \langle \text{Increment the line number and print a progress report at certain times 28} \rangle}
\]

\[
\text{\quad inp_desc-limit \leftarrow final_limit \leftarrow 0;}
\]

\[
\text{\quad while (inp_desc-limit < buf_size) \{}
\]

\[
\text{\quad \quad c \leftarrow fgetc(inp_desc-the_file);}
\]

\[
\text{\quad \quad \langle \text{Check c for EOF, return if line was empty, otherwise break to process last line 29} \rangle}
\]

\[
\text{\quad \quad inp_desc-buffer[inp_desc-limit++] \leftarrow c \leftarrow map_xord(c);}
\]

\[
\text{\quad \quad if (c \equiv nl_mark) break; \quad \text{\textit{end of line found \triangleleft}}}
\]

\[
\text{\quad \quad if (c \not\equiv @'\n' \land c \not\equiv \tab_mark \land c \not\equiv @'\r') final_limit \leftarrow inp_desc-limit;}
\]

\[
\text{\quad \}}
\]

\[
\text{\quad \langle \text{Test for truncated line, skip to end of line 30} \rangle}
\]

\[
\text{\quad inp_desc-limit \leftarrow final_limit;}
\]

\[
\text{\}}
\]

This code is used in section 24*.
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

\[ \text{err\_print}(!\_\text{Error\_message}\_\text{(file\_no)}) ; \]

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.

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

(31*) Error handling functions

```c
static void err_loc(int i) { prints location of error <
    printf("\_\text{(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.

```c
#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.

```c
#define jump_out() exit(EXIT\_FAILURE)
```
Handling multiple change files.

In the standard version we take the name of the files from the command line. It is assumed that filenames

34* 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.

(Prepare the change files 37*)

```
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*)

```
```
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)
    {
            return (true);
    }
    return (false);
The function $e_{of\_ch\_preamble}$ returns true if the input line from file $i$ starts with @y.

```java
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);
}
```

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.

```java
#define none (max_file_index + 1)
(Scan all other files for changes to be done 47*)
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.
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*)

\[
\begin{align*}
\text{if} \ (prod\_chf \equiv chf) \ \{ \\
& \langle \text{Test for normal, break when done 49} \rangle \\
& \langle \text{Test for pre, break when done 50} \rangle \\
& \langle \text{Test for post, break when done 51} \rangle \\
\} \\
\text{else} \\
& \text{if} \ (test\_input \equiv none) \ put\_line(actual\_input);
\end{align*}
\]

This code is used in section 45.

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

(Internal functions 24*)

\[
\begin{align*}
\text{static void usage(void)} \\
& \{ \\
& \phantom{\text{static void usage(void)}} \text{print("Usage: tie -m|--c outfile master changefile(s)"; term_new_line; jump_out(); }
\}
\]

We must scan through the list of parameters, given in \textit{argv}. The number is in \textit{argc}. We must pay attention to the flag parameter. We need at least 5 parameters and can handle up to \textit{max_file_index} change files. The names of the file parameters will be inserted into the structure of \textit{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 \textit{input_organization} structure (index 0). As long as the number of files found (counted in \textit{no_ch}) is \(-1\) we have not yet found the output file name.

(Scan the parameters 56*)

\[
\begin{align*}
\{ \\
& \text{int act_arg; } \\
& \text{if} \ (argc < 5 \lor argc > max\_file\_index + 4 - 1) \ usage(); \\
& \text{no_ch} \leftarrow -1; \ \triangleright \text{fill this part of input\_organization <} \\
& \text{for} \ (act\_arg \leftarrow 1; act\_arg < argc; act\_arg++) \{ \\
& \ \text{if} \ (argv[act\_arg][0] \equiv \text{"--\}} \ \langle \text{Set a flag 57} \rangle \\
& \ \text{else} \ \langle \text{Get a file name 58} \rangle \\
& \} \\
& \text{if} \ (no\_ch \leq 0 \lor prod\_chf \equiv unknown) \ usage(); \\
\}
\]

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
\texttt{kpse\_web\_format} when opening input files, which triggers the inspection of the \texttt{WEBINPUTS} environment
variable. Secondly, we set \texttt{kpse\_program\_name} to ‘tie’. This means if the variable \texttt{WEBINPUTS.tie} is
present in \texttt{texmf.cnf} (or \texttt{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 \texttt{WEBINPUTS} (overridden by \texttt{WEBINPUTS\_tie});
(b) a line in KPATHSEA configuration file \texttt{texmf.cnf},
e.g., \texttt{WEBINPUTS=$TEXMFDOTDIR:$TEXMF/texmf/web//}
or \texttt{WEBINPUTS\_tie=$TEXMFDOTDIR:$TEXMF/texmf/web//}.

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

\begin{verbatim}
⟨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); \triangleright print a "banner line" \triangleleft
printLn(versionstring); \triangleright Web2C version \triangleleft
printLn(copyright); \triangleright include the copyright notice \triangleleft
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_nI("(%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: \triangleright Anything except spotless, troublesome, or fatal is a bug. \triangleleft
  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*.
\end{verbatim}
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, 31, 32, 33, 34, 36, 37, 38, 39, 42, 43, 44, 47, 48, 55, 56, 59, 60, 62.

.idx: 21

act_arg: 56* 57, 58.
actual_input: 22, 45, 46, 47, 48, 50, 51, 52, 53, 55*
arg: 56* 59*
argv: 56* 57, 58, 59*
ASCII_Code: 7, 9, 21, 40, 42*
At the end of change file...: 43*
b: 39*
banner: 1* 59*
boolean: 4* 26, 38*, 39*, 43*, 44*
buf_size: 5*, 20, 21, 27*
buffer: 21, 24*, 27*, 38*, 40, 42*, 43*, 44*
buffer_index: 20, 21, 38*, 42*
c: 27*, 40.
Change file ended...: 41.
Change file entry...: 54.

err_loc: 31*, 47*
error_loc: 31*,
exit: 33*, 60*
EXIT_FAILURE: 33*, 60*
EXIT_SUCCESS: 60*
fatal: 6*, 32*, 60*
feof: 24*
fflush: 17.
file_no: 31*
file_types: 18*, 21, 22.
final_limit: 27*
first_text_char: 8, 14.
fopen: 34*
form_feed: 11, 13.
fprintf: 15*, 31*, 32*, 60*.
get_line: 24*, 36*, 40, 41, 52.
history: 6*, 31*, 32*, 60*
in_fileModes: 18*, 21.
init_change_file: 37*, 39*, 46, 47*
inp_desc: 24*, 25, 27*, 28, 29, 30, 39*, 40, 41,
43*, 44*, 46*, 58.
Input line too long: 30.
inout_has_ended: 26, 29, 45, 53.
inout_organization: 23, 24*, 31*, 36*, 37*, 38*, 39*, 42*
43*, 44*, 45, 46, 47*, 50, 51, 52, 54, 56*, 58.
j: 38*, 42*
jump_out: 32*, 33*, 55*
k: 38*
kpse_open_file: 36*, 37*
kpse_program_name: 59*
kpse_set_program_name: 59*
kpse_web_format: 36*, 37*, 59*
last_text_char: 8, 9, 14.
lines_dont_match: 38*, 47*
lmt: 38*, 42*
loop: 3*, 40, 48*
maint: 59*
malloc: 16*, 58.
map_xchr: 9, 42*, 49, 50, 51, 53.
map_xord: 9, 27*, 30.
max_ASCII: 7, 9.
max_file_index: 5*, 20, 23, 47*, 56*
new_line: 15*, 31*, 32*, 42*, 49, 50, 51, 53, 60*
No memory for descriptor: 58.
one: 47*, 48*, 49, 50, 51, 52.
normal: 19*, 49, 51, 59*
ord: 9.
out_md_type: 19*, 22.
out_mode: 22, 49, 50, 51, 53, 59*
out_name: 34*, 35, 58.
§62 TIE

INDEX 13

p: 42*
post: 19* 50, 51, 53.
pre: 19* 49, 50.
print: 15* 55* 59*
print_c: 15* 28.
printLn: 15* 59*
print_nl: 15* 43*
print2: 15* 28, 36* 37* 43*
print2_ln: 15*
print2_nl: 15* 60*
print3: 15*
print3_ln: 15* 31*
prod_chf: 22, 48* 56* 57.
put_line: 42, 48* 50, 51.
reading: 18* 46, 47* 52.
return_code: 6*
search: 18* 46, 47* 58.
Sections do not match: 47*
spotless: 6* 60*
stderr: 15* 31* 32* 60*
stdin: 15*
stdout: 15*
string: 4* 21, 35, 59*
system dependencies: 5* 7, 8, 9, 13, 15* 16*
   17, 29, 60* 61.
tab character expansion: 24* 27*
tab_mark: 11, 13, 27*
term_new_line: 15* 36* 37* 43* 55* 60*
term_out: 15* 17.
test: 18* 47*
test_file: 45, 47*
test_input: 22, 47* 48* 49, 50, 51, 52.
text_char: 8, 9.
text_file: 8, 21, 35.
the_file: 21, 24* 27* 30, 36* 37*
This can’t happen….: 46.
tieusage: 15*
troublesome: 6* 31* 60*
true: 4* 25, 29, 37* 38* 43* 44* 46, 47* 52.
unknown: 18* 22, 56* 57.
update_terminal: 17, 28.
usage: 15* 55* 56* 57.
versionstring: 1* 59*
WEBINPUTS: 59*
Where is the match…: 40.
xchr: 9, 10, 12, 13, 14.
xord: 9, 12, 14.
(Check that all changes have been read) Used in section 59*.
(Check the current files for any ends of changes) Used in section 45.
(Check c for EOF, return if line was empty, otherwise break to process last line) Used in section 27*.
(Error handling functions) Used in section 2*.
(Get a file name) Used in section 56*.
(Get line into buffer) Used in section 24*.
(Get the master file started) Used in section 59*.
(Global #includes) Used in section 2*.
(Global constants) Used in section 2*.
(Global types) Used in section 2*.
(Global variables) Used in section 2*.
(Handle end of file and return) Used in section 24*.
(Handle output) Used in section 45.
(Increment the line number and print a progress report at certain times) Used in section 27*.
(Local variables for initialisation) Used in section 59*.
(Prepare the change files) Used in section 59*.
(Prepare the output file) Used in section 59*.
(Print the job history) Used in section 59*.
(Process a line, break when end of source reached) Used in section 53.
(Process the input) Used in section 59*.
(Scan all other files for changes to be done) Used in section 45.
(Scan the parameters) Used in section 59*.
(Set a flag) Used in section 56*.
(Set initial values) Used in section 59*.
(Skip over comment lines; return if end of file) Used in section 39*.
(Skip to the next nonblank line; return if end of file) Used in section 39*.
(Step to next line) Used in section 45.
(Test for normal, break when done) Used in section 49*.
(Test for post, break when done) Used in section 48*.
(Test for pre, break when done) Used in section 48*.
(Test for truncated line, skip to end of line) Used in section 27*.
(The main function) Used in section 2*.