The TIE processor

(CWEB Version 2.4 [TeX Live])

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>The character set</td>
<td>7</td>
</tr>
<tr>
<td>Input and output</td>
<td>15</td>
</tr>
<tr>
<td>Data structures</td>
<td>18</td>
</tr>
<tr>
<td>File I/O</td>
<td>24</td>
</tr>
<tr>
<td>Reporting errors to the user</td>
<td>31</td>
</tr>
<tr>
<td>Handling multiple change files</td>
<td>34</td>
</tr>
<tr>
<td>Input/output organization</td>
<td>38</td>
</tr>
<tr>
<td>The main program</td>
<td>59</td>
</tr>
<tr>
<td>System-dependent changes</td>
<td>61</td>
</tr>
<tr>
<td>Index</td>
<td>62</td>
</tr>
</tbody>
</table>

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 verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

June 11, 2023 at 13:15
§1  TIE

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.

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*.
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*. 
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_in(v)
    {  
        fprintf(term_out, "%s", v);  term_new_line;
    }  // ‘print’ and then start new line
#define print2_in(a, b)
    {
        print2(a, b);  term_new_line;
    }  // same with two arguments
#define print3_in(a, b, c)
    {
        print3(a, b, c);  new_line(stderr);
    }  // same with three arguments
#define print_in(v)
    {  
        term_new_line;  print(v);
    }  // print information starting on a new line
#define print2_in(a, b)
    {
        term_new_line;  print2(a, b);
    }  // same for two arguments
```

(Global `#includes`) ≡

```c
#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 \texttt{in\_file\_modes}.

The mode \texttt{search} indicates that \texttt{TIE} searches for a match of the input line with any line of an input file in \texttt{reading} mode. \texttt{test} is used whenever a match is found and it has to be tested if the next input lines do match also. \texttt{reading} describes that the lines can be read without any check for matching other lines. \texttt{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.

\texttt{file\_types} is used to describe whether a file is a master file or a change file. The value \texttt{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.)

\begin{verbatim}
typedef enum {
  search, test, reading, ignore
} in_file_modes;

typedef enum {
  unknown, master, chf
} file_types;
\end{verbatim}

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

\begin{verbatim}
typedef enum {
  normal, pre, post
} out_md_type;
\end{verbatim}
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.

\[\text{Internal functions } 24^*\] ≡
\[
\text{static void get_line(file_index \(i\))} \\
\{ \\
\text{register input_description \(*inp\_desc \leftarrow input\_organization[i];\) } \\
\text{if (\(inp\_desc\_mode \equiv ignore\)) return; } \\
\text{if (feof(inp_desc-the_file))} \langle\text{Handle end of file and return } 25\rangle \\
\langle\text{Get line into buffer } 27^*\rangle \\
\}
\]

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.

\[\text{Get line into buffer } 27^*\] ≡
\[
\{ \\
\text{int final\_limit;} \quad \triangleright \text{used to delete trailing spaces } \triangleleft \\
\text{int } c; \quad \triangleright \text{the actual character read } \triangleleft \\
\langle\text{Increment the line number and print a progress report at certain times } 28\rangle \\
\text{inp\_desc\_limit } \leftarrow \text{final\_limit } \leftarrow 0; \\
\text{while (inp\_desc\_limit } \lt \text{ buf\_size}) \{ \\
\text{c } \leftarrow \text{fgetc(inp\_desc-the_file);} \\
\langle\text{Check } c \text{ for EOF, return if line was empty, otherwise break to process last line } 29^*\rangle \\
\text{inp\_desc\_buffer[inp\_desc\_limit++ ] } \leftarrow c \leftarrow \text{map\_xord(c);} \\
\text{if (c } \equiv \text{nLmark) break; } \quad \triangleright \text{end of line found } \triangleleft \\
\text{if (c } \neq \text{\texttt{\textbackslash t}} \wedge c \neq \text{tab\_mark } \wedge c \neq \text{\texttt{\textbackslash r}}) \text{final\_limit } \leftarrow \text{inp\_desc\_limit}; \\
\} \\
\langle\text{Test for truncated line, skip to end of line } 30\rangle \\
\text{inp\_desc\_limit } \leftarrow \text{final\_limit}; \\
\}
\]

This code is used in section 24*. 
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.

\[
\text{Check } c \text{ for EOF, return if line was empty, otherwise break to process last line} \quad 29^* \equiv \\
\text{if } (c \equiv \text{EOF}) \{ \\
\text{if } (\text{ inp_desc-limit } \leq 0) \{ \\
\text{ inp_desc-mode } \leftarrow \text{ ignore}; \text{ inp_desc-limit } \leftarrow -1; \quad \triangleright \text{ mark end-of-file } \quad \triangleleft \\
\text{ if } (\text{ inp_desc-type_of_file } \equiv \text{ master}) \text{ input_has Ended } \leftarrow \text{ true}; \text{ return; } \\
\} \\
\text{ else } \{ \quad \triangleright \text{ add end of line mark } \quad \triangleleft \\
\text{ c } \leftarrow \text{ nl_mark}; \text{ break; } \\
\} \\
\}
\]

This code is used in section 27^*.
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("!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); err_loc

(Error handling functions 31*)

static void err_loc(int i) ▷ prints location of error «
{
    print3 Ln("(file_%s, _line)\n", input_organization[i]-name_of_file, input_organization[i]-line);
}
```

This code is used in section 2*.

Non recoverable errors are handled by calling fatalError 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. fatalError cannot state any files because the problem is usually to access these.

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

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)
```
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.

```plaintext
(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*.

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

```plaintext
(Get the master file started 36*) ≡
{
  input_organization[0]←kpse_open_file(input_organization[0]−name_of_file,kpse_web_format);
  if (input_organization[0]−the_file ≡ Λ) fatalError("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*.

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`.

```plaintext
(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) fatalError("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  TIE

**Input/output organization.**

Here's a simple function that checks if two lines are different.

```java
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);
}
```

**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.

```java
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⟩
    ⟨Skip to the next nonblank line; return if end of file⟩
}
```

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

```java
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);
}
```

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

```java
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("%!a", input_organization[i]-name_of_file); term_new_line; return (true);
    } else if (inp_desc-limit ≥ 2)
        if (inp_desc-buffer[0] ≡ 0’0’ ∧ (inp_desc-buffer[1] ≡ 0’Z’ ∨ inp_desc-buffer[1] ≡ 0’z’))
            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*)

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.
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.

\[
\text{Handle output } 48^* \equiv \\
\text{if } (\text{prod.chf} \equiv \text{chf}) \text{ loop } \{ \\
\text{Test for normal, break when done } 49 \} \\
\text{Test for pre, break when done } 50 \} \\
\text{Test for post, break when done } 51 \} \\
\text{else} \\
\text{if } (\text{test.input} \equiv \text{none}) \text{ put.line(actual.input);} \\
\text{This code is used in section } 45.\]

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.

\[
\text{Process the input } 53^* \equiv \\
\text{actual.input} \leftarrow 0; \text{ input.has._ended } \leftarrow \text{false}; \\
\text{while } (\text{input.has._ended} \equiv \text{false} \lor \text{actual.input} \neq 0) \\
\text{Process a line, break when end of source reached } 45 \} \\
\text{if } (\text{out.mode} \equiv \text{pre}) \{ \quad \triangleright \text{last line has been deleted } \triangleleft \\
\text{fputc(map.xchr('@'@'), out.file); fputc(map.xchr('@'y'), out.file); new_line(out.file);} \\
\text{out.mode} \leftarrow \text{post}; \\
\} \\
\text{if } (\text{out.mode} \equiv \text{post}) \{ \quad \triangleright \text{last line has been changed } \triangleleft \\
\text{fputc(map.xchr('@'@'), out.file); fputc(map.xchr('@'z'), out.file); new_line(out.file);} \\
\} \\
\text{This code is used in section } 59^*.
\]

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.

\[
\text{Internal functions } 24^* \\n\text{static void usage(void)} \\
\{ \\
\text{print("Usage: tie[[-|m]-c outfile master changefile(s)]"); term.new.line; jump_out();} \\
\}
\]
We must scan through the list of parameters, given in `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.

```c
⟨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*. 

\[56^*\]
§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*)

```c
int main(int argc, string *argv)
{

    // Local variables for initialisation
    // Set initial values

    kpse_set_program_name(argv[0], "tie");  // print a “banner line”
    print(banner);  // Web2C version
    print(ln(versionstring));  // include the copyright notice
    actual_input ← 0;  // out_mode ← normal;  // Scan the parameters
    // Prepare the output file
    // Get the master file started
    // Prepare the change files
    // Check that all changes have been read
    // Print the job history

    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*)

```
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, 17, 18, 19, 21, 27, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39, 42, 43, 44, 47, 48, 53, 55, 56, 59, 60, 62.

_idsc: 21.*
act_arg: 56* 57, 58.
actual_input: 22, 45, 46, 47* 48* 50, 51, 52, 53* 59*.
argy: 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.
chf: 18* 48* 51, 57, 58.
chr: 9.
copyright: 1* 59*.
Could not open change file: 37*.
Could not open master file: 36*.
Could not open/create output file: 34*.
decr: 15* 46.
do_nothing: 3* 14, 30, 47*.
e_of_ch_module: 43* 46.
e_of_ch_preamble: 44* 52.
EOF: 27* 29* 30.
err_loc: 31* 47*.
err_print: 30, 31* 32* 40, 41, 47* 54.
error_loc: 31*.
exit: 33* 60*.
EXIT_FAILURE: 33* 60*.
EXIT_SUCCESS: 60*.
false: 4* 26, 38* 43* 44* 47* 53*.
fatal: 6* 32* 60*.
fatal_error: 32* 34* 36* 37* 46, 58.
feof: 24*.
fflush: 17.
fgets: 27* 30.
file_index: 20, 22, 24* 37* 38* 39* 42* 43* 44* 45, 54.
file_no: 31*.
file_types: 18* 21, 22.
final_limit: 27*.
first_text_char: 8, 14.
ffopen: 34*.
form_feed: 11, 13.
fprintf: 15* 31* 32* 60*.
ftpuse: 15* 32* 42* 49, 50, 51, 53*.
get_line: 24* 30* 40, 41, 52.
history: 6* 31* 32* 60*.
i: 12, 24* 31* 37* 38* 39* 42* 43* 44* 54.
ignore: 18* 24* 25, 29* 40, 41, 47* 54.
in_file_modes: 18* 21.
incr: 15* 28, 37* 47* 58.
init_change_file: 37* 39* 46, 47*.
Input line too long: 30.
input_description: 21, 23, 24* 39* 43* 44* 46, 58.
input_has Ended: 25, 26, 29* 45, 53*.
input_organisation: 23, 24* 31* 36* 37* 38* 39* 42* 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*.
lst_text_char: 8, 9, 14.
limit: 21, 24* 25, 27* 29* 38* 40, 41, 42* 43* 44* 45, 58.
line: 21, 24* 28, 31* 58.
lines_dont_match: 38* 47*.
lint: 38* 42*.
loop: 3* 40, 48*.
mall: 59*.
malloc: 16* 58.
map_xch: 9, 42* 49, 50, 51, 53*.
map_xord: 9, 27* 30.
master: 18* 25, 28, 29* 36* 46, 50, 57.
maxASCII: 7, 9.
max_file_index: 5* 20, 23, 47* 56*.
mode: 21, 24* 25, 29* 40, 41, 46, 47* 52, 54, 58.
name_of_file: 21, 31* 36* 37* 43* 58.
new_line: 15* 31* 32* 42* 49, 50, 51, 53* 60*.
null_mark: 11, 13, 27* 29* 30.
No memory for descriptor: 58.
no_ch: 22, 37* 47* 54, 56* 58.
one: 47* 48* 49, 50, 51, 52.
normal: 19* 49, 51, 59*.
ord: 9.
out_file: 34* 35, 42* 49, 50, 51, 53*.
out_md_type: 19* 22.
out_mode: 22, 49, 50, 51, 53* 59*.
out_name: 34* 35, 58.
p: 42*
post: 19* 50, 51, 53*
pre: 19* 49, 50, 53*
print: 15* 55* 59*
print_c: 15* 28.
print_Ln: 15* 59*
print_nI: 15* 43*
print2: 15* 28, 36* 37* 43*
print2_Ln: 15*
print2_nI: 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 29*.
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*.
Internal functions
Used in section 2*.
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 48*.
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*.