Linux Cross Reference

Free Electrons

Embedded Linux Experts

• *source navigation* • diff markup • identifier search • freetext search •

Version:
2.0.40 2.2.26 2.4.37 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 3.17 3.18 3.19 4.0 4.1 ***4.2***

Linux/scripts/unifdef.c

```
 1 /*
 2  * Copyright (c) 2002 - 2011 Tony Finch <dot@dotat.at>
 3  *
 4  * Redistribution and use in source and binary forms, with or without
 5  * modification, are permitted provided that the following conditions
 6  * are met:
 7  * 1. Redistributions of source code must retain the above copyright
 8  *    notice, this list of conditions and the following disclaimer.
 9  * 2. Redistributions in binary form must reproduce the above copyright
10  *    notice, this list of conditions and the following disclaimer in the
11  *    documentation and/or other materials provided with the distribution.
12  *
13  * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
14  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
15  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
16  * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
17  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
18  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
19  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
20  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
21  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
22  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
23  * SUCH DAMAGE.
24  */
25
26 /*
27  * unifdef - remove ifdef'ed lines
28  *
29  * This code was derived from software contributed to Berkeley by Dave Yost.
30  * It was rewritten to support ANSI C by Tony Finch. The original version
31  * of unifdef carried the 4-clause BSD copyright licence. None of its code
32  * remains in this version (though some of the names remain) so it now
33  * carries a more liberal licence.
34  *
35  *  Wishlist:
36  *      provide an option which will append the name of the
37  *        appropriate symbol after #else's and #endif's
38  *      provide an option which will check symbols after
39  *        #else's and #endif's to see that they match their
40  *        corresponding #ifdef or #ifndef
41  *
42  *   These require better buffer handling, which would also make
43  *   it possible to handle all "dodgy" directives correctly.
```

```
 */

#include <sys/types.h>
#include <sys/stat.h>

#include <ctype.h>
#include <err.h>
#include <errno.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

const char copyright[] =
    "@(#) $Version: unifdef-2.5 $\n"
    "@(#) $Author: Tony Finch (dot@dotat.at) $\n"
    "@(#) $URL: http://dotat.at/prog/unifdef $\n"
;

/* types of input lines: */
typedef enum {
        LT_TRUEI,               /* a true #if with ignore flag */
        LT_FALSEI,              /* a false #if with ignore flag */
        LT_IF,                  /* an unknown #if */
        LT_TRUE,                /* a true #if */
        LT_FALSE,               /* a false #if */
        LT_ELIF,                /* an unknown #elif */
        LT_ELTRUE,              /* a true #elif */
        LT_ELFALSE,             /* a false #elif */
        LT_ELSE,                /* #else */
        LT_ENDIF,               /* #endif */
        LT_DODGY,               /* flag: directive is not on one line */
        LT_DODGY_LAST = LT_DODGY + LT_ENDIF,
        LT_PLAIN,               /* ordinary line */
        LT_EOF,                 /* end of file */
        LT_ERROR,               /* unevaluable #if */
        LT_COUNT
} Linetype;

static char const * const linetype_name[] = {
        "TRUEI", "FALSEI", "IF", "TRUE", "FALSE",
        "ELIF", "ELTRUE", "ELFALSE", "ELSE", "ENDIF",
        "DODGY TRUEI", "DODGY FALSEI",
        "DODGY IF", "DODGY TRUE", "DODGY FALSE",
        "DODGY ELIF", "DODGY ELTRUE", "DODGY ELFALSE",
        "DODGY ELSE", "DODGY ENDIF",
        "PLAIN", "EOF", "ERROR"
};

/* state of #if processing */
typedef enum {
        IS_OUTSIDE,
        IS_FALSE_PREFIX,        /* false #if followed by false #elifs */
        IS_TRUE_PREFIX,         /* first non-false #(el)if is true */
        IS_PASS_MIDDLE,         /* first non-false #(el)if is unknown */
        IS_FALSE_MIDDLE,        /* a false #elif after a pass state */
        IS_TRUE_MIDDLE,         /* a true #elif after a pass state */
        IS_PASS_ELSE,           /* an else after a pass state */
        IS_FALSE_ELSE,          /* an else after a true state */
        IS_TRUE_ELSE,           /* an else after only false states */
        IS_FALSE_TRAILER,       /* #elifs after a true are false */
        IS_COUNT
} Ifstate;
```

```
109
110 static char const * const ifstate_name[] = {
111         "OUTSIDE", "FALSE_PREFIX", "TRUE_PREFIX",
112         "PASS_MIDDLE", "FALSE_MIDDLE", "TRUE_MIDDLE",
113         "PASS_ELSE", "FALSE_ELSE", "TRUE_ELSE",
114         "FALSE_TRAILER"
115 };
116
117 /* state of comment parser */
118 typedef enum {
119         NO_COMMENT = false,     /* outside a comment */
120         C_COMMENT,              /* in a comment like this one */
121         CXX_COMMENT,            /* between // and end of line */
122         STARTING_COMMENT,       /* just after slash-backslash-newline */
123         FINISHING_COMMENT,      /* star-backslash-newline in a C comment */
124         CHAR_LITERAL,           /* inside '' */
125         STRING_LITERAL          /* inside "" */
126 } Comment_state;
127
128 static char const * const comment_name[] = {
129         "NO", "C", "CXX", "STARTING", "FINISHING", "CHAR", "STRING"
130 };
131
132 /* state of preprocessor line parser */
133 typedef enum {
134         LS_START,               /* only space and comments on this line */
135         LS_HASH,                /* only space, comments, and a hash */
136         LS_DIRTY                /* this line can't be a preprocessor line */
137 } Line_state;
138
139 static char const * const linestate_name[] = {
140         "START", "HASH", "DIRTY"
141 };
142
143 /*
144  * Minimum translation limits from ISO/IEC 9899:1999 5.2.4.1
145  */
146 #define MAXDEPTH        64                      /* maximum #if nesting */
147 #define MAXLINE         4096                    /* maximum length of line */
148 #define MAXSYMS         4096                    /* maximum number of symbols */
149
150 /*
151  * Sometimes when editing a keyword the replacement text is longer, so
152  * we leave some space at the end of the tline buffer to accommodate this.
153  */
154 #define EDITSLOP        10
155
156 /*
157  * For temporary filenames
158  */
159 #define TEMPLATE        "unifdef.XXXXXX"
160
161 /*
162  * Globals.
163  */
164
165 static bool             compblank;              /* -B: compress blank lines */
166 static bool             lnblank;                /* -b: blank deleted lines */
167 static bool             complement;             /* -c: do the complement */
168 static bool             debugging;              /* -d: debugging reports */
169 static bool             iocccok;                /* -e: fewer IOCCC errors */
170 static bool             strictlogic;            /* -K: keep ambiguous #ifs */
171 static bool             killconsts;             /* -k: eval constant #ifs */
172 static bool             lnnum;                  /* -n: add #line directives */
173 static bool             symlist;                /* -s: output symbol list */
```

```
174 static bool             symdepth;               /* -S: output symbol depth */
175 static bool             text;                   /* -t: this is a text file */
176
177 static const char       *symname[MAXSYMS];       /* symbol name */
178 static const char       *value[MAXSYMS];         /* -Dsym=value */
179 static bool             ignore[MAXSYMS];        /* -iDsym or -iUsym */
180 static int              nsyms;                  /* number of symbols */
181
182 static FILE             *input;                 /* input file pointer */
183 static const char       *filename;              /* input file name */
184 static int              linenum;                /* current line number */
185 static FILE             *output;                /* output file pointer */
186 static const char       *ofilename;             /* output file name */
187 static bool             overwriting;            /* output overwrites input */
188 static char             tempname[FILENAME_MAX]; /* used when overwriting */
189
190 static char             tline[MAXLINE+EDITSLOP];/* input buffer plus space */
191 static char             *keyword;               /* used for editing #elif's */
192
193 static const char       *newline;               /* input file format */
194 static const char       newline_unix[] = "\n";
195 static const char       newline_crlf[] = "\r\n";
196
197 static Comment_state    incomment;              /* comment parser state */
198 static Line_state       linestate;              /* #if line parser state */
199 static Ifstate          ifstate[MAXDEPTH];      /* #if processor state */
200 static bool             ignoring[MAXDEPTH];     /* ignore comments state */
201 static int              stifline[MAXDEPTH];     /* start of current #if */
202 static int              depth;                  /* current #if nesting */
203 static int              delcount;               /* count of deleted lines */
204 static unsigned         blankcount;             /* count of blank lines */
205 static unsigned         blankmax;               /* maximum recent blankcount */
206 static bool             constexpr;              /* constant #if expression */
207 static bool             zerosyms = true;        /* to format symdepth output */
208 static bool             firstsym;               /* ditto */
209
210 static int              exitstat;               /* program exit status */
211
212 static void             addsym(bool, bool, char *);
213 static void             closeout(void);
214 static void             debug(const char *, ...);
215 static void             done(void);
216 static void             error(const char *);
217 static int              findsym(const char *);
218 static void             flushline(bool);
219 static Linetype         parseline(void);
220 static Linetype         ifeval(const char **);
221 static void             ignoreoff(void);
222 static void             ignoreon(void);
223 static void             keywordedit(const char *);
224 static void             nest(void);
225 static void             process(void);
226 static const char       *skipargs(const char *);
227 static const char       *skipcomment(const char *);
228 static const char       *skipsym(const char *);
229 static void             state(Ifstate);
230 static int              strlcmp(const char *, const char *, size_t);
231 static void             unnest(void);
232 static void             usage(void);
233 static void             version(void);
234
235 #define endsym(c) (!isalnum((unsigned char)c) && c != '_')
236
237 /*
238  * The main program.
```

```
239  */
240 int
241 main(int argc, char *argv[])
242 {
243         int opt;
244
245         while ((opt = getopt(argc, argv, "i:D:U:I:o:bBcdeKklnsStV")) != -1)
246                 switch (opt) {
247                 case 'i': /* treat stuff controlled by these symbols as text */
248                         /*
249                          * For strict backwards-compatibility the U or D
250                          * should be immediately after the -i but it doesn't
251                          * matter much if we relax that requirement.
252                          */
253                         opt = *optarg++;
254                         if (opt == 'D')
255                                 addsym(true, true, optarg);
256                         else if (opt == 'U')
257                                 addsym(true, false, optarg);
258                         else
259                                 usage();
260                         break;
261                 case 'D': /* define a symbol */
262                         addsym(false, true, optarg);
263                         break;
264                 case 'U': /* undef a symbol */
265                         addsym(false, false, optarg);
266                         break;
267                 case 'I': /* no-op for compatibility with cpp */
268                         break;
269                 case 'b': /* blank deleted lines instead of omitting them */
270                 case 'l': /* backwards compatibility */
271                         lnblank = true;
272                         break;
273                 case 'B': /* compress blank lines around removed section */
274                         compblank = true;
275                         break;
276                 case 'c': /* treat -D as -U and vice versa */
277                         complement = true;
278                         break;
279                 case 'd':
280                         debugging = true;
281                         break;
282                 case 'e': /* fewer errors from dodgy lines */
283                         iocccok = true;
284                         break;
285                 case 'K': /* keep ambiguous #ifs */
286                         strictlogic = true;
287                         break;
288                 case 'k': /* process constant #ifs */
289                         killconsts = true;
290                         break;
291                 case 'n': /* add #line directive after deleted lines */
292                         lnnum = true;
293                         break;
294                 case 'o': /* output to a file */
295                         ofilename = optarg;
296                         break;
297                 case 's': /* only output list of symbols that control #ifs */
298                         symlist = true;
299                         break;
300                 case 'S': /* list symbols with their nesting depth */
301                         symlist = symdepth = true;
302                         break;
303                 case 't': /* don't parse C comments */
```

```
304                         text = true;
305                         break;
306                 case 'V': /* print version */
307                         version();
308                 default:
309                         usage();
310                 }
311         argc -= optind;
312         argv += optind;
313         if (compblank && lnblank)
314                 errx(2, "-B and -b are mutually exclusive");
315         if (argc > 1) {
316                 errx(2, "can only do one file");
317         } else if (argc == 1 && strcmp(*argv, "-") != 0) {
318                 filename = *argv;
319                 input = fopen(filename, "rb");
320                 if (input == NULL)
321                         err(2, "can't open %s", filename);
322         } else {
323                 filename = "[stdin]";
324                 input = stdin;
325         }
326         if (ofilename == NULL) {
327                 ofilename = "[stdout]";
328                 output = stdout;
329         } else {
330                 struct stat ist, ost;
331                 if (stat(ofilename, &ost) == 0 &&
332                     fstat(fileno(input), &ist) == 0)
333                         overwriting = (ist.st_dev == ost.st_dev
334                                        && ist.st_ino == ost.st_ino);
335                 if (overwriting) {
336                         const char *dirsep;
337                         int ofd;
338
339                         dirsep = strrchr(ofilename, '/');
340                         if (dirsep != NULL)
341                                 snprintf(tempname, sizeof(tempname),
342                                     "%.*s/" TEMPLATE,
343                                     (int)(dirsep - ofilename), ofilename);
344                         else
345                                 snprintf(tempname, sizeof(tempname),
346                                     TEMPLATE);
347                         ofd = mkstemp(tempname);
348                         if (ofd != -1)
349                                 output = fdopen(ofd, "wb+");
350                         if (output == NULL)
351                                 err(2, "can't create temporary file");
352                         fchmod(ofd, ist.st_mode & (S_IRWXU|S_IRWXG|S_IRWXO));
353                 } else {
354                         output = fopen(ofilename, "wb");
355                         if (output == NULL)
356                                 err(2, "can't open %s", ofilename);
357                 }
358         }
359         process();
360         abort(); /* bug */
361 }
362
363 static void
364 version(void)
365 {
366         const char *c = copyright;
367         for (;;) {
368                 while (*++c != '$')
```

```
369                         if (*c == '\0')
370                                 exit(0);
371                 while (*++c != '$')
372                         putc(*c, stderr);
373                 putc('\n', stderr);
374         }
375 }
376
377 static void
378 usage(void)
379 {
380         fprintf(stderr, "usage: unifdef [-bBcdeKknsStV] [-Ipath]"
381             " [-Dsym[=val]] [-Usym] [-iDsym[=val]] [-iUsym] ... [file]\n");
382         exit(2);
383 }
384
385 /*
386  * A state transition function alters the global #if processing state
387  * in a particular way. The table below is indexed by the current
388  * processing state and the type of the current line.
389  *
390  * Nesting is handled by keeping a stack of states; some transition
391  * functions increase or decrease the depth. They also maintain the
392  * ignore state on a stack. In some complicated cases they have to
393  * alter the preprocessor directive, as follows.
394  *
395  * When we have processed a group that starts off with a known-false
396  * #if/#elif sequence (which has therefore been deleted) followed by a
397  * #elif that we don't understand and therefore must keep, we edit the
398  * latter into a #if to keep the nesting correct. We use strncpy() to
399  * overwrite the 4 byte token "elif" with "if  " without a '\0' byte.
400  *
401  * When we find a true #elif in a group, the following block will
402  * always be kept and the rest of the sequence after the next #elif or
403  * #else will be discarded. We edit the #elif into a #else and the
404  * following directive to #endif since this has the desired behaviour.
405  *
406  * "Dodgy" directives are split across multiple lines, the most common
407  * example being a multi-line comment hanging off the right of the
408  * directive. We can handle them correctly only if there is no change
409  * from printing to dropping (or vice versa) caused by that directive.
410  * If the directive is the first of a group we have a choice between
411  * failing with an error, or passing it through unchanged instead of
412  * evaluating it. The latter is not the default to avoid questions from
413  * users about unifdef unexpectedly leaving behind preprocessor directives.
414  */
415 typedef void state_fn(void);
416
417 /* report an error */
418 static void Eelif (void) { error("Inappropriate #elif"); }
419 static void Eelse (void) { error("Inappropriate #else"); }
420 static void Eendif(void) { error("Inappropriate #endif"); }
421 static void Eeof  (void) { error("Premature EOF"); }
422 static void Eioccc(void) { error("Obfuscated preprocessor control line"); }
423 /* plain line handling */
424 static void print (void) { flushline(true); }
425 static void drop  (void) { flushline(false); }
426 /* output lacks group's start line */
427 static void Strue (void) { drop();  ignoreoff(); state(IS_TRUE_PREFIX); }
428 static void Sfalse(void) { drop();  ignoreoff(); state(IS_FALSE_PREFIX); }
429 static void Selse (void) { drop();               state(IS_TRUE_ELSE); }
430 /* print/pass this block */
431 static void Pelif (void) { print(); ignoreoff(); state(IS_PASS_MIDDLE); }
432 static void Pelse (void) { print();              state(IS_PASS_ELSE); }
433 static void Pendif(void) { print(); unnest(); }
```

```
434 /* discard this block */
435 static void Dfalse(void) { drop();  ignoreoff(); state(IS_FALSE_TRAILER); }
436 static void Delif (void) { drop();  ignoreoff(); state(IS_FALSE_MIDDLE); }
437 static void Delse (void) { drop();               state(IS_FALSE_ELSE); }
438 static void Dendif(void) { drop();  unnest(); }
439 /* first line of group */
440 static void Fdrop (void) { nest();  Dfalse(); }
441 static void Fpass (void) { nest();  Pelif(); }
442 static void Ftrue (void) { nest();  Strue(); }
443 static void Ffalse(void) { nest();  Sfalse(); }
444 /* variable pedantry for obfuscated lines */
445 static void Oiffy (void) { if (!iocccok) Eioccc(); Fpass(); ignoreon(); }
446 static void Oif   (void) { if (!iocccok) Eioccc(); Fpass(); }
447 static void Oelif (void) { if (!iocccok) Eioccc(); Pelif(); }
448 /* ignore comments in this block */
449 static void Idrop (void) { Fdrop();  ignoreon(); }
450 static void Itrue (void) { Ftrue();  ignoreon(); }
451 static void Ifalse(void) { Ffalse(); ignoreon(); }
452 /* modify this line */
453 static void Mpass (void) { strncpy(keyword, "if  ", 4); Pelif(); }
454 static void Mtrue (void) { keywordedit("else");  state(IS_TRUE_MIDDLE); }
455 static void Melif (void) { keywordedit("endif"); state(IS_FALSE_TRAILER); }
456 static void Melse (void) { keywordedit("endif"); state(IS_FALSE_ELSE); }
457
458 static state_fn * const trans_table[IS_COUNT][LT_COUNT] = {
459 /* IS_OUTSIDE */
460 { Itrue, Ifalse,Fpass, Ftrue, Ffalse,Eelif, Eelif, Eelif, Eelse, Eendif,
461   Oiffy, Oiffy, Fpass, Oif,   Oif,   Eelif, Eelif, Eelif, Eelse, Eendif,
462   print, done,  abort },
463 /* IS_FALSE_PREFIX */
464 { Idrop, Idrop, Fdrop, Fdrop, Fdrop, Mpass, Strue, Sfalse,Selse, Dendif,
465   Idrop, Idrop, Fdrop, Fdrop, Fdrop, Mpass, Eioccc,Eioccc,Eioccc,Eioccc,
466   drop,  Eeof,  abort },
467 /* IS_TRUE_PREFIX */
468 { Itrue, Ifalse,Fpass, Ftrue, Ffalse,Dfalse,Dfalse,Dfalse,Delse, Dendif,
469   Oiffy, Oiffy, Fpass, Oif,   Oif,   Eioccc,Eioccc,Eioccc,Eioccc,Eioccc,
470   print, Eeof,  abort },
471 /* IS_PASS_MIDDLE */
472 { Itrue, Ifalse,Fpass, Ftrue, Ffalse,Pelif, Mtrue, Delif, Pelse, Pendif,
473   Oiffy, Oiffy, Fpass, Oif,   Oif,   Pelif, Oelif, Oelif, Pelse, Pendif,
474   print, Eeof,  abort },
475 /* IS_FALSE_MIDDLE */
476 { Idrop, Idrop, Fdrop, Fdrop, Fdrop, Pelif, Mtrue, Delif, Pelse, Pendif,
477   Idrop, Idrop, Fdrop, Fdrop, Fdrop, Eioccc,Eioccc,Eioccc,Eioccc,Eioccc,
478   drop,  Eeof,  abort },
479 /* IS_TRUE_MIDDLE */
480 { Itrue, Ifalse,Fpass, Ftrue, Ffalse,Melif, Melif, Melif, Melse, Pendif,
481   Oiffy, Oiffy, Fpass, Oif,   Oif,   Eioccc,Eioccc,Eioccc,Eioccc,Pendif,
482   print, Eeof,  abort },
483 /* IS_PASS_ELSE */
484 { Itrue, Ifalse,Fpass, Ftrue, Ffalse,Eelif, Eelif, Eelif, Eelse, Pendif,
485   Oiffy, Oiffy, Fpass, Oif,   Oif,   Eelif, Eelif, Eelif, Eelse, Pendif,
486   print, Eeof,  abort },
487 /* IS_FALSE_ELSE */
488 { Idrop, Idrop, Fdrop, Fdrop, Fdrop, Eelif, Eelif, Eelif, Eelse, Dendif,
489   Idrop, Idrop, Fdrop, Fdrop, Fdrop, Eelif, Eelif, Eelif, Eelse, Eioccc,
490   drop,  Eeof,  abort },
491 /* IS_TRUE_ELSE */
492 { Itrue, Ifalse,Fpass, Ftrue, Ffalse,Eelif, Eelif, Eelif, Eelse, Dendif,
493   Oiffy, Oiffy, Fpass, Oif,   Oif,   Eelif, Eelif, Eelif, Eelse, Eioccc,
494   print, Eeof,  abort },
495 /* IS_FALSE_TRAILER */
496 { Idrop, Idrop, Fdrop, Fdrop, Fdrop, Dfalse,Dfalse,Dfalse,Delse, Dendif,
497   Idrop, Idrop, Fdrop, Fdrop, Fdrop, Dfalse,Dfalse,Dfalse,Delse, Eioccc,
498   drop,  Eeof,  abort }
```

```
499 /*TRUEI  FALSEI IF     TRUE   FALSE  ELIF   ELTRUE ELFALSE ELSE  ENDIF
500   TRUEI  FALSEI IF     TRUE   FALSE  ELIF   ELTRUE ELFALSE ELSE  ENDIF (DODGY)
501   PLAIN  EOF    ERROR */
502 };
503
504 /*
505  * State machine utility functions
506  */
507 static void
508 ignoreoff(void)
509 {
510         if (depth == 0)
511                 abort(); /* bug */
512         ignoring[depth] = ignoring[depth-1];
513 }
514 static void
515 ignoreon(void)
516 {
517         ignoring[depth] = true;
518 }
519 static void
520 keywordedit(const char *replacement)
521 {
522         snprintf(keyword, tline + sizeof(tline) - keyword,
523             "%s%s", replacement, newline);
524         print();
525 }
526 static void
527 nest(void)
528 {
529         if (depth > MAXDEPTH-1)
530                 abort(); /* bug */
531         if (depth == MAXDEPTH-1)
532                 error("Too many levels of nesting");
533         depth += 1;
534         stifline[depth] = linenum;
535 }
536 static void
537 unnest(void)
538 {
539         if (depth == 0)
540                 abort(); /* bug */
541         depth -= 1;
542 }
543 static void
544 state(Ifstate is)
545 {
546         ifstate[depth] = is;
547 }
548
549 /*
550  * Write a line to the output or not, according to command line options.
551  */
552 static void
553 flushline(bool keep)
554 {
555         if (symlist)
556                 return;
557         if (keep ^ complement) {
558                 bool blankline = tline[strspn(tline, " \t\r\n")] == '\0';
559                 if (blankline && compblank && blankcount != blankmax) {
560                         delcount += 1;
561                         blankcount += 1;
562                 } else {
563                         if (lnnum && delcount > 0)
```

```
564                                 printf("#line %d%s", linenum, newline);
565                         fputs(tline, output);
566                         delcount = 0;
567                         blankmax = blankcount = blankline ? blankcount + 1 : 0;
568                 }
569         } else {
570                 if (lnblank)
571                         fputs(newline, output);
572                 exitstat = 1;
573                 delcount += 1;
574                 blankcount = 0;
575         }
576         if (debugging)
577                 fflush(output);
578 }
579
580 /*
581  * The driver for the state machine.
582  */
583 static void
584 process(void)
585 {
586         /* When compressing blank lines, act as if the file
587            is preceded by a large number of blank lines. */
588         blankmax = blankcount = 1000;
589         for (;;) {
590                 Linetype lineval = parseline();
591                 trans_table[ifstate[depth]][lineval]();
592                 debug("process line %d %s -> %s depth %d",
593                     linenum, linetype_name[lineval],
594                     ifstate_name[ifstate[depth]], depth);
595         }
596 }
597
598 /*
599  * Flush the output and handle errors.
600  */
601 static void
602 closeout(void)
603 {
604         if (symdepth && !zerosyms)
605                 printf("\n");
606         if (fclose(output) == EOF) {
607                 warn("couldn't write to %s", ofilename);
608                 if (overwriting) {
609                         unlink(tempname);
610                         errx(2, "%s unchanged", filename);
611                 } else {
612                         exit(2);
613                 }
614         }
615 }
616
617 /*
618  * Clean up and exit.
619  */
620 static void
621 done(void)
622 {
623         if (incomment)
624                 error("EOF in comment");
625         closeout();
626         if (overwriting && rename(tempname, ofilename) == -1) {
627                 warn("couldn't rename temporary file");
628                 unlink(tempname);
```

```
629                 errx(2, "%s unchanged", ofilename);
630         }
631         exit(exitstat);
632 }
633
634 /*
635  * Parse a line and determine its type. We keep the preprocessor line
636  * parser state between calls in the global variable linestate, with
637  * help from skipcomment().
638  */
639 static Linetype
640 parseline(void)
641 {
642         const char *cp;
643         int cursym;
644         int kwlen;
645         Linetype retval;
646         Comment_state wascomment;
647
648         linenum++;
649         if (fgets(tline, MAXLINE, input) == NULL)
650                 return (LT_EOF);
651         if (newline == NULL) {
652                 if (strrchr(tline, '\n') == strrchr(tline, '\r') + 1)
653                         newline = newline_crlf;
654                 else
655                         newline = newline_unix;
656         }
657         retval = LT_PLAIN;
658         wascomment = incomment;
659         cp = skipcomment(tline);
660         if (linestate == LS_START) {
661                 if (*cp == '#') {
662                         linestate = LS_HASH;
663                         firstsym = true;
664                         cp = skipcomment(cp + 1);
665                 } else if (*cp != '\0')
666                         linestate = LS_DIRTY;
667         }
668         if (!incomment && linestate == LS_HASH) {
669                 keyword = tline + (cp - tline);
670                 cp = skipsym(cp);
671                 kwlen = cp - keyword;
672                 /* no way can we deal with a continuation inside a keyword */
673                 if (strncmp(cp, "\\\r\n", 3) == 0 ||
674                     strncmp(cp, "\\\n", 2) == 0)
675                         Eioccc();
676                 if (strlcmp("ifdef", keyword, kwlen) == 0 ||
677                     strlcmp("ifndef", keyword, kwlen) == 0) {
678                         cp = skipcomment(cp);
679                         if ((cursym = findsym(cp)) < 0)
680                                 retval = LT_IF;
681                         else {
682                                 retval = (keyword[2] == 'n')
683                                     ? LT_FALSE : LT_TRUE;
684                                 if (value[cursym] == NULL)
685                                         retval = (retval == LT_TRUE)
686                                             ? LT_FALSE : LT_TRUE;
687                                 if (ignore[cursym])
688                                         retval = (retval == LT_TRUE)
689                                             ? LT_TRUEI : LT_FALSEI;
690                         }
691                         cp = skipsym(cp);
692                 } else if (strlcmp("if", keyword, kwlen) == 0)
693                         retval = ifeval(&cp);
```

```
694                 else if (strlcmp("elif", keyword, kwlen) == 0)
695                         retval = ifeval(&cp) - LT_IF + LT_ELIF;
696                 else if (strlcmp("else", keyword, kwlen) == 0)
697                         retval = LT_ELSE;
698                 else if (strlcmp("endif", keyword, kwlen) == 0)
699                         retval = LT_ENDIF;
700                 else {
701                         linestate = LS_DIRTY;
702                         retval = LT_PLAIN;
703                 }
704                 cp = skipcomment(cp);
705                 if (*cp != '\0') {
706                         linestate = LS_DIRTY;
707                         if (retval == LT_TRUE || retval == LT_FALSE ||
708                             retval == LT_TRUEI || retval == LT_FALSEI)
709                                 retval = LT_IF;
710                         if (retval == LT_ELTRUE || retval == LT_ELFALSE)
711                                 retval = LT_ELIF;
712                 }
713                 if (retval != LT_PLAIN && (wascomment || incomment)) {
714                         retval += LT_DODGY;
715                         if (incomment)
716                                 linestate = LS_DIRTY;
717                 }
718                 /* skipcomment normally changes the state, except
719                    if the last line of the file lacks a newline, or
720                    if there is too much whitespace in a directive */
721                 if (linestate == LS_HASH) {
722                         size_t len = cp - tline;
723                         if (fgets(tline + len, MAXLINE - len, input) == NULL) {
724                                 /* append the missing newline */
725                                 strcpy(tline + len, newline);
726                                 cp += strlen(newline);
727                                 linestate = LS_START;
728                         } else {
729                                 linestate = LS_DIRTY;
730                         }
731                 }
732         }
733         if (linestate == LS_DIRTY) {
734                 while (*cp != '\0')
735                         cp = skipcomment(cp + 1);
736         }
737         debug("parser line %d state %s comment %s line", linenum,
738             comment_name[incomment], linestate_name[linestate]);
739         return (retval);
740 }
741
742 /*
743  * These are the binary operators that are supported by the expression
744  * evaluator.
745  */
746 static Linetype op_strict(int *p, int v, Linetype at, Linetype bt) {
747         if(at == LT_IF || bt == LT_IF) return (LT_IF);
748         return (*p = v, v ? LT_TRUE : LT_FALSE);
749 }
750 static Linetype op_lt(int *p, Linetype at, int a, Linetype bt, int b) {
751         return op_strict(p, a < b, at, bt);
752 }
753 static Linetype op_gt(int *p, Linetype at, int a, Linetype bt, int b) {
754         return op_strict(p, a > b, at, bt);
755 }
756 static Linetype op_le(int *p, Linetype at, int a, Linetype bt, int b) {
757         return op_strict(p, a <= b, at, bt);
758 }
```

```
759 static Linetype op_ge(int *p, Linetype at, int a, Linetype bt, int b) {
760         return op_strict(p, a >= b, at, bt);
761 }
762 static Linetype op_eq(int *p, Linetype at, int a, Linetype bt, int b) {
763         return op_strict(p, a == b, at, bt);
764 }
765 static Linetype op_ne(int *p, Linetype at, int a, Linetype bt, int b) {
766         return op_strict(p, a != b, at, bt);
767 }
768 static Linetype op_or(int *p, Linetype at, int a, Linetype bt, int b) {
769         if (!strictlogic && (at == LT_TRUE || bt == LT_TRUE))
770                 return (*p = 1, LT_TRUE);
771         return op_strict(p, a || b, at, bt);
772 }
773 static Linetype op_and(int *p, Linetype at, int a, Linetype bt, int b) {
774         if (!strictlogic && (at == LT_FALSE || bt == LT_FALSE))
775                 return (*p = 0, LT_FALSE);
776         return op_strict(p, a && b, at, bt);
777 }
778
779 /*
780  * An evaluation function takes three arguments, as follows: (1) a pointer to
781  * an element of the precedence table which lists the operators at the current
782  * level of precedence; (2) a pointer to an integer which will receive the
783  * value of the expression; and (3) a pointer to a char* that points to the
784  * expression to be evaluated and that is updated to the end of the expression
785  * when evaluation is complete. The function returns LT_FALSE if the value of
786  * the expression is zero, LT_TRUE if it is non-zero, LT_IF if the expression
787  * depends on an unknown symbol, or LT_ERROR if there is a parse failure.
788  */
789 struct ops;
790
791 typedef Linetype eval_fn(const struct ops *, int *, const char **);
792
793 static eval_fn eval_table, eval_unary;
794
795 /*
796  * The precedence table. Expressions involving binary operators are evaluated
797  * in a table-driven way by eval_table. When it evaluates a subexpression it
798  * calls the inner function with its first argument pointing to the next
799  * element of the table. Innermost expressions have special non-table-driven
800  * handling.
801  */
802 static const struct ops {
803         eval_fn *inner;
804         struct op {
805                 const char *str;
806                 Linetype (*fn)(int *, Linetype, int, Linetype, int);
807         } op[5];
808 } eval_ops[] = {
809         { eval_table, { { "||", op_or } } },
810         { eval_table, { { "&&", op_and } } },
811         { eval_table, { { "==", op_eq },
812                         { "!=", op_ne } } },
813         { eval_unary, { { "<=", op_le },
814                         { ">=", op_ge },
815                         { "<", op_lt },
816                         { ">", op_gt } } }
817 };
818
819 /*
820  * Function for evaluating the innermost parts of expressions,
821  * viz. !expr (expr) number defined(symbol) symbol
822  * We reset the constexpr flag in the last two cases.
823  */
```

```
824 static Linetype
825 eval_unary(const struct ops *ops, int *valp, const char **cpp)
826 {
827         const char *cp;
828         char *ep;
829         int sym;
830         bool defparen;
831         Linetype lt;
832
833         cp = skipcomment(*cpp);
834         if (*cp == '!') {
835                 debug("eval%d !", ops - eval_ops);
836                 cp++;
837                 lt = eval_unary(ops, valp, &cp);
838                 if (lt == LT_ERROR)
839                         return (LT_ERROR);
840                 if (lt != LT_IF) {
841                         *valp = !*valp;
842                         lt = *valp ? LT_TRUE : LT_FALSE;
843                 }
844         } else if (*cp == '(') {
845                 cp++;
846                 debug("eval%d (", ops - eval_ops);
847                 lt = eval_table(eval_ops, valp, &cp);
848                 if (lt == LT_ERROR)
849                         return (LT_ERROR);
850                 cp = skipcomment(cp);
851                 if (*cp++ != ')')
852                         return (LT_ERROR);
853         } else if (isdigit((unsigned char)*cp)) {
854                 debug("eval%d number", ops - eval_ops);
855                 *valp = strtol(cp, &ep, 0);
856                 if (ep == cp)
857                         return (LT_ERROR);
858                 lt = *valp ? LT_TRUE : LT_FALSE;
859                 cp = skipsym(cp);
860         } else if (strncmp(cp, "defined", 7) == 0 && endsym(cp[7])) {
861                 cp = skipcomment(cp+7);
862                 debug("eval%d defined", ops - eval_ops);
863                 if (*cp == '(') {
864                         cp = skipcomment(cp+1);
865                         defparen = true;
866                 } else {
867                         defparen = false;
868                 }
869                 sym = findsym(cp);
870                 if (sym < 0) {
871                         lt = LT_IF;
872                 } else {
873                         *valp = (value[sym] != NULL);
874                         lt = *valp ? LT_TRUE : LT_FALSE;
875                 }
876                 cp = skipsym(cp);
877                 cp = skipcomment(cp);
878                 if (defparen && *cp++ != ')')
879                         return (LT_ERROR);
880                 constexpr = false;
881         } else if (!endsym(*cp)) {
882                 debug("eval%d symbol", ops - eval_ops);
883                 sym = findsym(cp);
884                 cp = skipsym(cp);
885                 if (sym < 0) {
886                         lt = LT_IF;
887                         cp = skipargs(cp);
888                 } else if (value[sym] == NULL) {
```

```
889                         *valp = 0;
890                         lt = LT_FALSE;
891                 } else {
892                         *valp = strtol(value[sym], &ep, 0);
893                         if (*ep != '\0' || ep == value[sym])
894                                 return (LT_ERROR);
895                         lt = *valp ? LT_TRUE : LT_FALSE;
896                         cp = skipargs(cp);
897                 }
898                 constexpr = false;
899         } else {
900                 debug("eval%d bad expr", ops - eval_ops);
901                 return (LT_ERROR);
902         }
903
904         *cpp = cp;
905         debug("eval%d = %d", ops - eval_ops, *valp);
906         return (lt);
907 }
908
909 /*
910  * Table-driven evaluation of binary operators.
911  */
912 static Linetype
913 eval_table(const struct ops *ops, int *valp, const char **cpp)
914 {
915         const struct op *op;
916         const char *cp;
917         int val;
918         Linetype lt, rt;
919
920         debug("eval%d", ops - eval_ops);
921         cp = *cpp;
922         lt = ops->inner(ops+1, valp, &cp);
923         if (lt == LT_ERROR)
924                 return (LT_ERROR);
925         for (;;) {
926                 cp = skipcomment(cp);
927                 for (op = ops->op; op->str != NULL; op++)
928                         if (strncmp(cp, op->str, strlen(op->str)) == 0)
929                                 break;
930                 if (op->str == NULL)
931                         break;
932                 cp += strlen(op->str);
933                 debug("eval%d %s", ops - eval_ops, op->str);
934                 rt = ops->inner(ops+1, &val, &cp);
935                 if (rt == LT_ERROR)
936                         return (LT_ERROR);
937                 lt = op->fn(valp, lt, *valp, rt, val);
938         }
939
940         *cpp = cp;
941         debug("eval%d = %d", ops - eval_ops, *valp);
942         debug("eval%d lt = %s", ops - eval_ops, linetype_name[lt]);
943         return (lt);
944 }
945
946 /*
947  * Evaluate the expression on a #if or #elif line. If we can work out
948  * the result we return LT_TRUE or LT_FALSE accordingly, otherwise we
949  * return just a generic LT_IF.
950  */
951 static Linetype
952 ifeval(const char **cpp)
953 {
```

```
954         int ret;
955         int val = 0;
956
957         debug("eval %s", *cpp);
958         constexpr = killconsts ? false : true;
959         ret = eval_table(eval_ops, &val, cpp);
960         debug("eval = %d", val);
961         return (constexpr ? LT_IF : ret == LT_ERROR ? LT_IF : ret);
962 }
963
964 /*
965  * Skip over comments, strings, and character literals and stop at the
966  * next character position that is not whitespace. Between calls we keep
967  * the comment state in the global variable incomment, and we also adjust
968  * the global variable linestate when we see a newline.
969  * XXX: doesn't cope with the buffer splitting inside a state transition.
970  */
971 static const char *
972 skipcomment(const char *cp)
973 {
974         if (text || ignoring[depth]) {
975                 for (; isspace((unsigned char)*cp); cp++)
976                         if (*cp == '\n')
977                                 linestate = LS_START;
978                 return (cp);
979         }
980         while (*cp != '\0')
981                 /* don't reset to LS_START after a line continuation */
982                 if (strncmp(cp, "\\\r\n", 3) == 0)
983                         cp += 3;
984                 else if (strncmp(cp, "\\\n", 2) == 0)
985                         cp += 2;
986                 else switch (incomment) {
987                 case NO_COMMENT:
988                         if (strncmp(cp, "/\\\r\n", 4) == 0) {
989                                 incomment = STARTING_COMMENT;
990                                 cp += 4;
991                         } else if (strncmp(cp, "/\\\n", 3) == 0) {
992                                 incomment = STARTING_COMMENT;
993                                 cp += 3;
994                         } else if (strncmp(cp, "/*", 2) == 0) {
995                                 incomment = C_COMMENT;
996                                 cp += 2;
997                         } else if (strncmp(cp, "//", 2) == 0) {
998                                 incomment = CXX_COMMENT;
999                                 cp += 2;
1000                         } else if (strncmp(cp, "\'", 1) == 0) {
1001                                 incomment = CHAR_LITERAL;
1002                                 linestate = LS_DIRTY;
1003                                 cp += 1;
1004                         } else if (strncmp(cp, "\"", 1) == 0) {
1005                                 incomment = STRING_LITERAL;
1006                                 linestate = LS_DIRTY;
1007                                 cp += 1;
1008                         } else if (strncmp(cp, "\n", 1) == 0) {
1009                                 linestate = LS_START;
1010                                 cp += 1;
1011                         } else if (strchr(" \r\t", *cp) != NULL) {
1012                                 cp += 1;
1013                         } else
1014                                 return (cp);
1015                         continue;
1016                 case CXX_COMMENT:
1017                         if (strncmp(cp, "\n", 1) == 0) {
1018                                 incomment = NO_COMMENT;
```

```
1019                                    linestate = LS_START;
1020                            }
1021                            cp += 1;
1022                            continue;
1023                    case CHAR_LITERAL:
1024                    case STRING_LITERAL:
1025                            if ((incomment == CHAR_LITERAL && cp[0] == '\'') ||
1026                                (incomment == STRING_LITERAL && cp[0] == '\"')) {
1027                                    incomment = NO_COMMENT;
1028                                    cp += 1;
1029                            } else if (cp[0] == '\\') {
1030                                    if (cp[1] == '\0')
1031                                            cp += 1;
1032                                    else
1033                                            cp += 2;
1034                            } else if (strncmp(cp, "\n", 1) == 0) {
1035                                    if (incomment == CHAR_LITERAL)
1036                                            error("unterminated char literal");
1037                                    else
1038                                            error("unterminated string literal");
1039                            } else
1040                                    cp += 1;
1041                            continue;
1042                    case C_COMMENT:
1043                            if (strncmp(cp, "*\\\r\n", 4) == 0) {
1044                                    incomment = FINISHING_COMMENT;
1045                                    cp += 4;
1046                            } else if (strncmp(cp, "*\\\n", 3) == 0) {
1047                                    incomment = FINISHING_COMMENT;
1048                                    cp += 3;
1049                            } else if (strncmp(cp, "*/", 2) == 0) {
1050                                    incomment = NO_COMMENT;
1051                                    cp += 2;
1052                            } else
1053                                    cp += 1;
1054                            continue;
1055                    case STARTING_COMMENT:
1056                            if (*cp == '*') {
1057                                    incomment = C_COMMENT;
1058                                    cp += 1;
1059                            } else if (*cp == '/') {
1060                                    incomment = CXX_COMMENT;
1061                                    cp += 1;
1062                            } else {
1063                                    incomment = NO_COMMENT;
1064                                    linestate = LS_DIRTY;
1065                            }
1066                            continue;
1067                    case FINISHING_COMMENT:
1068                            if (*cp == '/') {
1069                                    incomment = NO_COMMENT;
1070                                    cp += 1;
1071                            } else
1072                                    incomment = C_COMMENT;
1073                            continue;
1074                    default:
1075                            abort(); /* bug */
1076                    }
1077            return (cp);
1078 }
1079
1080 /*
1081  * Skip macro arguments.
1082  */
1083 static const char *
```

```
1084 skipargs(const char *cp)
1085 {
1086         const char *ocp = cp;
1087         int level = 0;
1088         cp = skipcomment(cp);
1089         if (*cp != '(')
1090                 return (cp);
1091         do {
1092                 if (*cp == '(')
1093                         level++;
1094                 if (*cp == ')')
1095                         level--;
1096                 cp = skipcomment(cp+1);
1097         } while (level != 0 && *cp != '\0');
1098         if (level == 0)
1099                 return (cp);
1100         else
1101         /* Rewind and re-detect the syntax error later. */
1102                 return (ocp);
1103 }
1104
1105 /*
1106  * Skip over an identifier.
1107  */
1108 static const char *
1109 skipsym(const char *cp)
1110 {
1111         while (!endsym(*cp))
1112                 ++cp;
1113         return (cp);
1114 }
1115
1116 /*
1117  * Look for the symbol in the symbol table. If it is found, we return
1118  * the symbol table index, else we return -1.
1119  */
1120 static int
1121 findsym(const char *str)
1122 {
1123         const char *cp;
1124         int symind;
1125
1126         cp = skipsym(str);
1127         if (cp == str)
1128                 return (-1);
1129         if (symlist) {
1130                 if (symdepth && firstsym)
1131                         printf("%s%3d", zerosyms ? "" : "\n", depth);
1132                 firstsym = zerosyms = false;
1133                 printf("%s%.*s%s",
1134                     symdepth ? " " : "",
1135                     (int)(cp-str), str,
1136                     symdepth ? "" : "\n");
1137                 /* we don't care about the value of the symbol */
1138                 return (0);
1139         }
1140         for (symind = 0; symind < nsyms; ++symind) {
1141                 if (strlcmp(symname[symind], str, cp-str) == 0) {
1142                         debug("findsym %s %s", symname[symind],
1143                             value[symind] ? value[symind] : "");
1144                         return (symind);
1145                 }
1146         }
1147         return (-1);
1148 }
```

```
1149
1150 /*
1151  * Add a symbol to the symbol table.
1152  */
1153 static void
1154 addsym(bool ignorethis, bool definethis, char *sym)
1155 {
1156         int symind;
1157         char *val;
1158
1159         symind = findsym(sym);
1160         if (symind < 0) {
1161                 if (nsyms >= MAXSYMS)
1162                         errx(2, "too many symbols");
1163                 symind = nsyms++;
1164         }
1165         symname[symind] = sym;
1166         ignore[symind] = ignorethis;
1167         val = sym + (skipsym(sym) - sym);
1168         if (definethis) {
1169                 if (*val == '=') {
1170                         value[symind] = val+1;
1171                         *val = '\0';
1172                 } else if (*val == '\0')
1173                         value[symind] = "1";
1174                 else
1175                         usage();
1176         } else {
1177                 if (*val != '\0')
1178                         usage();
1179                 value[symind] = NULL;
1180         }
1181         debug("addsym %s=%s", symname[symind],
1182             value[symind] ? value[symind] : "undef");
1183 }
1184
1185 /*
1186  * Compare s with n characters of t.
1187  * The same as strncmp() except that it checks that s[n] == '\0'.
1188  */
1189 static int
1190 strlcmp(const char *s, const char *t, size_t n)
1191 {
1192         while (n-- && *t != '\0')
1193                 if (*s != *t)
1194                         return ((unsigned char)*s - (unsigned char)*t);
1195                 else
1196                         ++s, ++t;
1197         return ((unsigned char)*s);
1198 }
1199
1200 /*
1201  * Diagnostics.
1202  */
1203 static void
1204 debug(const char *msg, ...)
1205 {
1206         va_list ap;
1207
1208         if (debugging) {
1209                 va_start(ap, msg);
1210                 vwarnx(msg, ap);
1211                 va_end(ap);
1212         }
1213 }
```

```
1214
1215 static void
1216 error(const char *msg)
1217 {
1218         if (depth == 0)
1219                 warnx("%s: %d: %s", filename, linenum, msg);
1220         else
1221                 warnx("%s: %d: %s (#if line %d depth %d)",
1222                     filename, linenum, msg, stifline[depth], depth);
1223         closeout();
1224         errx(2, "output may be truncated");
1225 }
1226
```