

```

*****
3050 Thu Apr  2 12:57:05 2015
new/usr/src/test/libc-tests/cfg/symbols/README
2nd round review feedback from rmustacc.
*****
1 #
2 # This file and its contents are supplied under the terms of the
3 # Common Development and Distribution License ("CDDL"), version 1.0.
4 # You may only use this file in accordance with the terms of version
5 # 1.0 of the CDDL.
6 #
7 # A full copy of the text of the CDDL should have accompanied this
8 # source.  A copy of the CDDL is also available via the Internet at
9 # http://www.illumos.org/license/CDDL.
10 #
11 #
12 #
13 # Copyright 2015 Garrett D'Amore <garrett@damore.org>
14 #
15 #
16 The configuration files in this directory are structured using the
17 syntax defined in the ../README file.  They make use of the compilation
18 environments declared in ../compilation.cfg, and are processed by the
19 symbols test.
20 #
21 We have organized the files by header file, that is the tests for symbols
22 declared in a header file (e.g. <unistd.h> appear in a file based on that
23 header file's name (e.g. unistd_h.cfg.)  This is purely for convenience.
24 #
25 Within these various declarations, we have the following field types:
26 #
27 <envs>   This is a list of compilation environments where the symbol
28           should be legal.  To indicate that the symbol must not be legal
29           an environment group can be prefixed with "-".  For example,
30           "SUS+ -SUSv4+" indicates a symbol that is legal in all SUS
31           "SUS -SUSv4+" indicates a symbol that is legal in all SUS
32           environments up to SUSv3, and was removed in SUSv4 and subsequent
33           versions of SUS.  As you can see, we can list multiple environments
34           or environment groups, and we can add or remove to previous groups
35           with subsequent ones.
36 #
37 <name>   This is a symbol name.  It follows the rules for C symbol names.
38 #
39 <header> This is a header file, for example, unistd.h.  Conventionally,
40           the header files used should match the file where the test is
41           declared.
42 #
43 <type>   This is a C type.  Function types can be declared without their
44           names, e.g. "void (*)(int)".  Structures (e.g. "struct stat") and
45           pointer types (e.g. "pthread_t *") are legal as well.
46 #
47 Here are the types of declarations in these files:
48 #
49 type | <name> | <header> | <envs>
50 #
51 Tests for a C type with <name>.  The test verifies that a variable with
52 this type can be declared when the <header> is included.
53 #
54 value | <name> | <type> | <header> | <envs>
55 #
56 Tests for a value named <name>, of type <type>.  The test attempts to
57 assign the given value to a scratch variable declared with the given
58 type.  The value can be a macro or other C symbol.
59 #
60 func | <name> | <type> | <type> [; <type> ]... | <header> | <envs>

```

```

61 Tests whether a function <name>, returning the first <type>, and
62 taking arguments of following <type> values, is declared.  Note that
63 the argument types are separated by semicolons.  For varargs style
64 functions, leave out the ... part.  For function declarations
65 that have no declared arguments, either void can specified, or
66 the type list can be omitted.

```

68 Examples:

```

70 type | size_t | sys/types.h | ALL
71 value | NULL | void * | stdlib.h | ALL
72 func strlen | int | const char *; int | string.h | ALL

```

```

*****
18221 Thu Apr  2 12:57:06 2015
new/usr/src/test/libc-tests/tests/symbols/symbols_test.c
2nd round review feedback from rmustacc.
*****
_____unchanged_portion_omitted_____

64 char *compiler = NULL;
65 const char *c89flags = NULL;
66 const char *c99flags = NULL;

68 #define MAXENV 64 /* maximum number of environments (bitmask width) */
69 #define MAXHDR 10 /* maximum # headers to require to access symbol */
70 #define MAXARG 20 /* maximum # of arguments */

72 #define WS " \t"

74 static int next_env = 0;

76 struct compile_env {
77     char *ce_name;
78     char *ce_lang;
79     char *ce_defs;
80     int ce_index;
77     char *name;
78     char *lang;
79     char *defs;
80     int index;
81 };

83 static struct compile_env compile_env[MAXENV];

85 struct env_group {
86     char *eg_name;
87     uint64_t eg_mask;
88     struct env_group *eg_next;
86     char *name;
87     uint64_t mask;
88     struct env_group *next;
89 };

91 typedef enum { SYM_TYPE, SYM_VALUE, SYM_FUNC } sym_type_t;

93 struct sym_test {
94     char *st_name;
95     sym_type_t st_type;
96     char *st_hdrs[MAXHDR];
97     char *st_rtype;
98     char *st_atypes[MAXARG];
99     uint64_t st_test_mask;
100    uint64_t st_need_mask;
101    char *st_prog;
102    struct sym_test *st_next;
94     char *name;
95     sym_type_t type;
96     char *hdrs[MAXHDR];
97     char *rtype;
98     char *atypes[MAXARG];
99     uint64_t test_mask;
100    uint64_t need_mask;
101    char *prog;
102    struct sym_test *next;
103 };
_____unchanged_portion_omitted_____

146 static void

```

```

147 append_sym_test(struct sym_test *st)
148 {
149     *sym_insert = st;
150     sym_insert = &st->st_next;
150     sym_insert = &st->next;
151 }

153 static int
154 find_env_mask(const char *name, uint64_t *mask)
155 {
156     for (int i = 0; i < MAXENV; i++) {
157         if (compile_env[i].ce_name != NULL &&
158             strcmp(compile_env[i].ce_name, name) == 0) {
156         for (int i = 0; i < 64; i++) {
157             if (compile_env[i].name != NULL &&
158                 strcmp(compile_env[i].name, name) == 0) {
159                 *mask |= (1ULL << i);
160                 return (0);
161             }
162         }

164         for (struct env_group *eg = env_groups; eg != NULL; eg = eg->eg_next) {
165             if (strcmp(name, eg->eg_name) == 0) {
166                 *mask |= eg->eg_mask;
164             for (struct env_group *eg = env_groups; eg != NULL; eg = eg->next) {
165                 if (strcmp(name, eg->name) == 0) {
166                     *mask |= eg->mask;
167                     return (0);
168                 }
169             }
170             return (-1);
171 }
_____unchanged_portion_omitted_____

227 static int
228 do_env(char **fields, int nfields, char **err)
229 {
230     char *name;
231     char *lang;
232     char *defs;

234     if (nfields != 3) {
235         myasprintf(err, "number of fields (%d) != 3", nfields);
236         return (-1);
237     }

239     if (next_env >= MAXENV) {
240         myasprintf(err, "too many environments");
241         return (-1);
242     }

244     name = fields[0];
245     lang = fields[1];
246     defs = fields[2];

248     compile_env[next_env].ce_name = mystrdup(name);
249     compile_env[next_env].ce_lang = mystrdup(lang);
250     compile_env[next_env].ce_defs = mystrdup(defs);
251     compile_env[next_env].ce_index = next_env;
248     compile_env[next_env].name = mystrdup(name);
249     compile_env[next_env].lang = mystrdup(lang);
250     compile_env[next_env].defs = mystrdup(defs);
251     compile_env[next_env].index = next_env;
252     next_env++;
253     return (0);
254 }

```

```

256 static int
257 do_env_group(char **fields, int nfields, char **err)
258 {
259     char *name;
260     char *list;
261     struct env_group *eg;
262     uint64_t mask;
263     char *item;

265     if (nfields != 2) {
266         myasprintf(err, "number of fields (%d) != 2", nfields);
267         return (-1);
268     }

270     name = fields[0];
271     list = fields[1];
272     mask = 0;

274     if (expand_env(list, &mask, &item) < 0) {
275         myasprintf(err, "reference to undefined env %s", item);
276         return (-1);
277     }

279     eg = myzalloc(sizeof (*eg));
280     eg->eg_name = mystrdup(name);
281     eg->eg_mask = mask;
282     eg->eg_next = env_groups;
280     eg->name = mystrdup(name);
281     eg->mask = mask;
282     eg->next = env_groups;
283     env_groups = eg;
284     return (0);
285 }

```

unchanged portion omitted

```

330 static void
331 mkprog(struct sym_test *st)
332 {
333     char *s;

335     proglen = 0;

337     for (int i = 0; i < MAXHDR && st->st_hdrs[i] != NULL; i++) {
338         addprogfmt("#include <%s>\n", st->st_hdrs[i]);
337     for (int i = 0; i < MAXHDR && st->hdrs[i] != NULL; i++) {
338         addprogfmt("#include <%s>\n", st->hdrs[i]);
339     }

341     for (s = st->st_rtype; *s; s++) {
341     for (s = st->rtype; *s; s++) {
342         addprogch(*s);
343         if (*s == '(') {
344             s++;
345             addprogch(*s);
346             s++;
347             break;
348         }
349     }
350     addprogch(' ');

352     /* for function pointers, s is closing suffix, otherwise empty */

354     switch (st->st_type) {
354     switch (st->type) {
355     case SYM_TYPE:

```

```

356         addprogstr("test_type;");
357         break;

359     case SYM_VALUE:
360         addprogfmt("test_value%s;\n", s); /* s usually empty */
361         addprogstr("void\ntest_func(void)\n{\n\n");
362         addprogfmt("\tttest_value = %s;\n", st->st_name);
362         addprogfmt("\tttest_value = %s;\n", st->name);
363         break;

365     case SYM_FUNC:
366         addprogstr("\ntest_func(");
367         for (int i = 0; st->st_atypes[i] != NULL && i < MAXARG; i++) {
367         for (int i = 0; st->atypes[i] != NULL && i < MAXARG; i++) {
368             int didname = 0;
369             if (i > 0) {
370                 addprogstr(", ");
371             }
372             if (strcmp(st->st_atypes[i], "void") == 0) {
372             if (strcmp(st->atypes[i], "void") == 0) {
373                 didname = 1;
374             }
375             if (strcmp(st->st_atypes[i], "") == 0) {
375             if (strcmp(st->atypes[i], "") == 0) {
376                 didname = 1;
377                 addprogstr("void");
378             }

380             /* print the argument list */
381             for (char *a = st->st_atypes[i]; *a; a++) {
381             for (char *a = st->atypes[i]; *a; a++) {
382                 if (*a == '(' && a[1] == '*' && !didname) {
382                 if (*a == '(' && a[1] == '*' && !didname) {
383                     addprogfmt("(*a%d", i);
383                     didname = 1;
384                     a++;
385                 } else if (*a == '[' && !didname) {
386                     addprogfmt("a%d", i);
387                     didname = 1;
388                 } else {
389                     addprogch(*a);
390                 }
391             }
392             if (!didname) {
393                 addprogfmt(" a%d", i);
394             }
395         }
396     }

398     if (st->st_atypes[0] == NULL) {
398     if (st->atypes[0] == NULL) {
399         addprogstr("void");
400     }

402     /*
403     * Close argument list, and closing ")" for func ptrs.
404     * Note that for non-function pointers, s will be empty
405     * below, otherwise it points to the trailing argument
406     * list.
407     */
408     addprogfmt(")%s\n{\n\n", s);
408     /* close argument list, and closing ")" for func ptrs */
409     addprogfmt(")%s\n{\n\n", s); /* NB: s is normally empty */

410     if (strcmp(st->st_rtype, "") != 0 &&
411         strcmp(st->st_rtype, "void") != 0) {
405     if (strcmp(st->rtype, "") != 0 &&
406         strcmp(st->rtype, "void") != 0) {

```

```

412         addprogstr("return ");
413     }

415     /* add the function call */
416     addprogfmt("%s(", st->st_name);
417     for (int i = 0; st->st_atypes[i] != NULL && i < MAXARG; i++) {
418         if (strcmp(st->st_atypes[i], "") != 0 &&
419             strcmp(st->st_atypes[i], "void") != 0) {
420             addprogfmt("%s(", st->name);
421             for (int i = 0; st->atypes[i] != NULL && i < MAXARG; i++) {
422                 if (strcmp(st->atypes[i], "") != 0 &&
423                     strcmp(st->atypes[i], "void") != 0) {
424                     addprogfmt("%sa%d", i > 0 ? ", " : "", i);
425                 }
426             }
427         }
428     }
429     addprogstr(");\n");
430     break;
431 }

432 addprogch('\n');

433 st->st_prog = progbuf;
434 st->prog = progbuf;
435 }

436 static int
437 add_envs(struct sym_test *st, char *envs, char **err)
438 {
439     char *item;
440     if (expand_env_list(envs, &st->st_test_mask, &st->st_need_mask,
441         &item) < 0) {
442         if (expand_env_list(envs, &st->test_mask, &st->need_mask, &item) < 0) {
443             myasprintf(err, "bad env action %s", item);
444             return (-1);
445         }
446     }
447     return (0);
448 }

449 static int
450 add_headers(struct sym_test *st, char *hdrs, char **err)
451 {
452     int i = 0;
453     for (char *h = strsep(&hdrs, ";"); h != NULL; h = strsep(&hdrs, ";")) {
454         if (i >= MAXHDR) {
455             myasprintf(err, "too many headers");
456             return (-1);
457         }
458         test_trim(&h);
459         st->st_hdrs[i++] = mystrdup(h);
460         st->hdrs[i++] = mystrdup(h);
461     }
462     return (0);
463 }

464 static int
465 add_arg_types(struct sym_test *st, char *atype, char **err)
466 {
467     int i = 0;
468     char *a;
469     for (a = strsep(&atype, ";"); a != NULL; a = strsep(&atype, ";")) {
470         if (i >= MAXARG) {
471             myasprintf(err, "too many arguments");
472             return (-1);
473         }
474     }

```

```

471     }
472     test_trim(&a);
473     st->st_atypes[i++] = mystrdup(a);
474     st->atypes[i++] = mystrdup(a);
475 }

476     return (0);
477 }

478 static int
479 do_type(char **fields, int nfields, char **err)
480 {
481     char *decl;
482     char *hdrs;
483     char *envs;
484     struct sym_test *st;
485
486     if (nfields != 3) {
487         myasprintf(err, "number of fields (%d) != 3", nfields);
488         return (-1);
489     }
490     decl = fields[0];
491     hdrs = fields[1];
492     envs = fields[2];
493
494     st = myzalloc(sizeof (*st));
495     st->st_type = SYM_TYPE;
496     st->st_name = mystrdup(decl);
497     st->st_rtype = mystrdup(decl);
498     st->type = SYM_TYPE;
499     st->name = mystrdup(decl);
500     st->rtype = mystrdup(decl);
501
502     if ((add_envs(st, envs, err) < 0) ||
503         (add_headers(st, hdrs, err) < 0)) {
504         return (-1);
505     }
506     append_sym_test(st);
507     return (0);
508 }

509 static int
510 do_value(char **fields, int nfields, char **err)
511 {
512     char *name;
513     char *type;
514     char *hdrs;
515     char *envs;
516     struct sym_test *st;
517
518     if (nfields != 4) {
519         myasprintf(err, "number of fields (%d) != 4", nfields);
520         return (-1);
521     }
522     name = fields[0];
523     type = fields[1];
524     hdrs = fields[2];
525     envs = fields[3];
526
527     st = myzalloc(sizeof (*st));
528     st->st_type = SYM_VALUE;
529     st->st_name = mystrdup(name);
530     st->st_rtype = mystrdup(type);
531     st->type = SYM_VALUE;
532     st->name = mystrdup(name);

```

```

524     st->rtype = mystrdup(type);

532     if ((add_envs(st, envs, err) < 0) ||
533         (add_headers(st, hdrs, err) < 0)) {
534         return (-1);
535     }
536     append_sym_test(st);

538     return (0);
539 }

541 static int
542 do_func(char **fields, int nfields, char **err)
543 {
544     char *name;
545     char *rtype;
546     char *atype;
547     char *hdrs;
548     char *envs;
549     struct sym_test *st;

551     if (nfields != 5) {
552         myasprintf(err, "number of fields (%d) != 5", nfields);
553         return (-1);
554     }
555     name = fields[0];
556     rtype = fields[1];
557     atype = fields[2];
558     hdrs = fields[3];
559     envs = fields[4];

561     st = myzalloc(sizeof (*st));
562     st->st_type = SYM_FUNC;
563     st->st_name = mystrdup(name);
564     st->st_rtype = mystrdup(rtype);
565     st->type = SYM_FUNC;
566     st->name = mystrdup(name);
567     st->rtype = mystrdup(rtype);

568     if ((add_envs(st, envs, err) < 0) ||
569         (add_headers(st, hdrs, err) < 0) ||
570         (add_arg_types(st, atype, err) < 0)) {
571         return (-1);
572     }
573     append_sym_test(st);

574     return (0);
575 }

576 struct sym_test *
577 next_sym_test(struct sym_test *st)
578 {
579     return (st == NULL ? sym_tests : st->st_next);
580     return (st == NULL ? sym_tests : st->next);
581 }

582 const char *
583 sym_test_prog(struct sym_test *st)
584 {
585     if (st->st_prog == NULL) {
586         if (st->prog == NULL) {
587             mkprog(st);
588         }
589     }
590     return (st->st_prog);
591     return (st->prog);
592 }

```

```

591 const char *
592 sym_test_name(struct sym_test *st)
593 {
594     return (st->st_name);
595     return (st->name);
596 }

597 /*
598 * Iterate through tests. Pass in NULL for cenv to begin the iteration. For
599 * subsequent iterations, use the return value from the previous iteration.
600 * Returns NULL when there are no more environments.
601 */
602 struct compile_env *
603 sym_test_env(struct sym_test *st, struct compile_env *cenv, int *need)
604 {
605     int i = cenv ? cenv->ce_index + 1 : 0;
606     int i = cenv ? cenv->index + 1 : 0;
607     uint64_t b = 1ULL << i;

608     while ((i < MAXENV) && (b != 0)) {
609         cenv = &compile_env[i];
610         if (b & st->st_test_mask) {
611             *need = (st->st_need_mask & b) ? 1 : 0;
612             if (b & st->test_mask) {
613                 *need = (st->need_mask & b) ? 1 : 0;
614                 return (cenv);
615             }
616             b <<= 1;
617             i++;
618         }
619         return (NULL);
620     }

621 const char *
622 env_name(struct compile_env *cenv)
623 {
624     return (cenv->ce_name);
625     return (cenv->name);
626 }

627 const char *
628 env_lang(struct compile_env *cenv)
629 {
630     return (cenv->ce_lang);
631     return (cenv->lang);
632 }

633 const char *
634 env_defs(struct compile_env *cenv)
635 {
636     return (cenv->ce_defs);
637     return (cenv->defs);
638 }

639 unchanged_portion_omitted

713 void
714 find_compiler(void)
715 {
716     test_t t;
717     int i;
718     FILE *cf;

720     t = test_start("finding compiler");

722     if ((cf = fopen(cfile, "w+")) == NULL) {

```

```

723         test_failed(t, "Unable to open %s for write: %s", cfile,
724                   strerror(errno));
725         return;
726     }
727     (void) fprintf(cf, "#include <stdio.h>\n");
728     (void) fprintf(cf, "int main(int argc, char **argv) {\n");
729     (void) fprintf(cf, "#if defined(__SUNPRO_C)\n");
730     (void) fprintf(cf, "exit(51);\n");
731     (void) fprintf(cf, "#elif defined(__GNUC__)\n");
732     (void) fprintf(cf, "exit(52);\n");
733     (void) fprintf(cf, "#else\n");
734     (void) fprintf(cf, "exit(99)\n");
735     (void) fprintf(cf, "#endif\n");
736     (void) fclose(cf);

738     for (i = 0; compilers[i] != NULL; i++) {
739         char cmd[256];
740         int rv;

742         (void) snprintf(cmd, sizeof (cmd),
743                        "%s %s %s -o %s >/dev/null 2>&1",
744                        compilers[i], MFLAG, cfile, efile);
745         test_debugf(t, "trying %s", cmd);
746         rv = system(cmd);

748         test_debugf(t, "result: %d", rv);

750         if ((rv < 0) || !WIFEXITED(rv) || WEXITSTATUS(rv) != 0)
751             continue;

753         rv = system(efile);
754         if (rv >= 0 && WIFEXITED(rv)) {
755             rv = WEXITSTATUS(rv);
756         } else {
757             rv = -1;
758         }

760         switch (rv) {
761             case 51: /* STUDIO */
762                 test_debugf(t, "Found Studio C");
763                 c89flags = "-Xc -errwarn=%all -v -xc99=%none " MFLAG;
764                 c99flags = "-Xc -errwarn=%all -v -xc99=%all " MFLAG;
765                 if (extra_debug) {
766                     test_debugf(t, "c89flags: %s", c89flags);
767                     test_debugf(t, "c99flags: %s", c99flags);
768                 }
769                 test_passed(t);
770                 break;
771             case 52: /* GCC */
772                 test_debugf(t, "Found GNU C");
773                 c89flags = "-Wall -Werror -std=c89 " MFLAG;
774                 c99flags = "-Wall -Werror -std=c99 " MFLAG;
775                 if (extra_debug) {
776                     test_debugf(t, "c89flags: %s", c89flags);
777                     test_debugf(t, "c99flags: %s", c99flags);
778                 }
779                 test_passed(t);
780                 break;
781             case 99:
782                 test_debugf(t, "Found unknown (unsupported) compiler");
783                 continue;
784             default:
785                 continue;
786         }
787         myasprintf(&compiler, "%s", compilers[i]);
788         test_debugf(t, "compiler: %s", compiler);

```

```

789         return;
790     }
791     test_failed(t, "No compiler found.");
792 }

```

unchanged portion omitted

```

865 void
866 test_compile(void)
867 {
868     struct sym_test *st;
869     struct compile_env *cenv;
870     test_t t;
871     int need;

873     for (st = next_sym_test(NULL); st; st = next_sym_test(st)) {
874         if ((sym != NULL) && strcmp(sym, sym_test_name(st))) {
875             continue;
876         }
877         /* XXX: we really want a sym_test_desc() */
878         for (cenv = sym_test_env(st, NULL, &need);
879              cenv != NULL;
880              cenv = sym_test_env(st, cenv, &need)) {
881             t = test_start("%s : %c%s", sym_test_name(st),
882                          t = test_start("%s : %c%s", st->name,
883                          need ? '+' : '-', env_name(cenv));
884             if (do_compile(t, st, cenv, need) == 0) {
885                 test_passed(t);
886             }
887         }

889         if (full_count > 0) {
890             test_summary();
891         }
892 }

```

unchanged portion omitted