

new/usr/src/cmd/file/Makefile

1

```
*****
2569 Fri Mar 1 17:09:59 2019
new/usr/src/cmd/file/Makefile
10476 file(1) could be smatch clean
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 # Copyright 2007 Sun Microsystems, Inc. All rights reserved.
23 # Use is subject to license terms.
24 #
25 # Copyright (c) 2018, Joyent, Inc.

27 PROG= file
28 XPG4PROG= file
29 MAGIC= magic

31 ELFCAP= $(SRC)/common/elfcap
32 SGRSRTCID= $(SRC)/common/sgsrtcid

34 LOBJS= file.o elf_read32.o elf_read64.o magicutils.o
35 OBJS= $(LOBJS) elfcap.o
36 XPG4OBJS= $(OBJS:%.o=xpg4_%.o)
37 SRCS= file.c elf_read.c magicutils.c $(ELFCAP)/elfcap.c

39 include ../Makefile.cmd

41 CSTD= $(CSTD_GNU99)
42 C99LMODE= -Xc99=%all

44 CERRWARN += -_gcc=-Wno-uninitialized
45 CERRWARN += -_gcc=-Wno-type-limits

47 # not linted
48 SMATCH=off

47 POFILE= file_all.po
48 POFILES= $(SRCS:%.c=%%.po)

50 # The debug binary can be built using the flags
51 # SOURCEDEBUG=yes CGLOBALSTATIC=
52 # This will avoid the multiple symbols definition error
53 # for static global variables in elf_read32.o and elf_read64.o

55 LDLIBS += -lelf
56 CPPFLAGS += -I$(ELFCAP) -I$(SGSRTCID)
57 $(XPG4) := CFLAGS += -DXPG4
```

new/usr/src/cmd/file/Makefile

2

```
59 ROOTETCMAGIC= $(MAGIC:%=$(ROOTETC)/%)
61 $(ROOTETCMAGIC) := FILEMODE = $(LIBFILEMODE)
63 .PARALLEL: $(OBJS) $(XPG4OBJS) $(POFILES)
65 .KEEP_STATE:
67 all: $(PROG) $(XPG4) $(MAGIC)
69 $(PROG) : $(OBJS)
70 $(LINK.c) $(OBJS) -o $@ $(LDLIBS)
71 $(POST_PROCESS)
73 $(XPG4) : $(XPG4OBJS)
74 $(LINK.c) $(XPG4OBJS) -o $@ $(LDLIBS)
75 $(POST_PROCESS)
77 %.o: %.c
78 $(COMPILE.c) -o $@ $<
80 %32.o: %.c
81 $(COMPILE.c) -o $@ $<
83 %64.o: %.c
84 $(COMPILE.c) -D_ELF64 -o $@ $<
86 xpg4_%.o: %.c
87 $(COMPILE.c) -o $@ $<
89 xpg4_%32.o: %.c
90 $(COMPILE.c) -o $@ $<
92 xpg4_%64.o: %.c
93 $(COMPILE.c) -D_ELF64 -o $@ $<
95 elfcap.o: $(ELFCAP)/elfcap.c
96 $(COMPILE.c) -o $@ $(ELFCAP)/elfcap.c
98 xpg4_elfcap.o: $(ELFCAP)/elfcap.c
99 $(COMPILE.c) -o $@ $(ELFCAP)/elfcap.c
101 $(POFILE): $(POFILES)
102 $(RM) $@
103 cat $(POFILES) > $@
105 install: all $(ROOTPROG) $(ROOTXPG4PROG) $(ROOTETCMAGIC)
107 clean:
108 $(RM) $(OBJS) $(XPG4OBJS)
110 lint: lint_SRCS
112 include ../Makefile.targ
```

```

*****
16042 Fri Mar 1 17:09:59 2019
new/usr/src/cmd/file/elf_read.c
10476 file(1) could be smatch clean
*****
_____unchanged_portion_omitted_____

408 /*
409 * process_shdr:      Read Section Headers to attempt to get HW/SW
410 *                   capabilities by looking at the SUNW_cap
411 *                   section and set string in Elf_Info.
412 *                   Also look for symbol tables and debug
413 *                   information sections. Set the "stripped" field
414 *                   in Elf_Info with corresponding flags.
415 */
416 static int
417 process_shdr(Elf_Info *EI)
418 {
419     int         mac;
420     int         i, j, idx;
421     char        *strtab;
422     size_t      strtab_sz;
423     Elf_Shdr    *shdr = &EI_Shdr;

425     mac = EI_Ehdr.e_machine;

427     /* if there are no sections, return success anyway */
428     if (EI_Ehdr.e_shoff == 0 && EI_Ehdr.shnum == 0)
429         return (ELF_READ_OKAY);

431     /* read section names from String Section */
432     if (get_shdr(EI, EI_Ehdr.shstrndx) == ELF_READ_FAIL)
433         return (ELF_READ_FAIL);

435     if ((strtab = malloc(shdr->sh_size)) == NULL)
436         return (ELF_READ_FAIL);

438     if (pread64(EI->elffd, strtab, shdr->sh_size, shdr->sh_offset)
439         != shdr->sh_size)
440         return (ELF_READ_FAIL);

442     strtab_sz = shdr->sh_size;

444     /* read all the sections and process them */
445     for (idx = 1, i = 0; i < EI_Ehdr.shnum; idx++, i++) {
446         char *shnam;

448         if (get_shdr(EI, i) == ELF_READ_FAIL)
449             return (ELF_READ_FAIL);

451         if (shdr->sh_type == SHT_NULL) {
452             idx--;
453             continue;
454         }

456         if (shdr->sh_type == SHT_SUNW_cap) {
457             char    capstr[128];
458             Elf_Cap  Chdr;
459             FILE_ELF_OFF_T cap_off;
460             FILE_ELF_SIZE_T csize;
461             int capn;

463             cap_off = shdr->sh_offset;
464             csize = sizeof (Elf_Cap);

466             if (shdr->sh_size == 0 || shdr->sh_entsize == 0) {

```

```

467         (void) fprintf(stderr, ELF_ERR_ELFCAP1,
468             File, EI->file);
469         return (ELF_READ_FAIL);
470     }
471     capn = (shdr->sh_size / shdr->sh_entsize);
472     for (j = 0; j < capn; j++) {
473         /*
474          * read cap and xlate the values
475          */
476         if ((pread64(EI->elffd, &Chdr, csize, cap_off)
477             != csize) ||
478             file_xlatetom(ELF_T_CAP, (char *)&Chdr)
479             == 0) {
480             (void) fprintf(stderr, ELF_ERR_ELFCAP2,
481                 File, EI->file);
482             return (ELF_READ_FAIL);
483         }

485         cap_off += csize;

487         /*
488          * Each capability group is terminated with
489          * CA_SUNW_NULL. Groups other than the first
490          * represent symbol capabilities, and aren't
491          * interesting here.
492          */
493         if (Chdr.c_tag == CA_SUNW_NULL)
494             break;

496         (void) elfcap_tag_to_str(ELFCAP_STYLE_UC,
497             Chdr.c_tag, Chdr.c_un.c_val, capstr,
498             sizeof (capstr), ELFCAP_FMT_SNGSPACE,
499             mac);

501         if ((*EI->cap_str != '\0') && (*capstr != '\0'))
502             (void) strlcat(EI->cap_str, " ",
503                 sizeof (EI->cap_str));

505         (void) strlcat(EI->cap_str, capstr,
506             sizeof (EI->cap_str));
507     }
508 } else if (shdr->sh_type == SHT_DYNAMIC) {
509     Elf_Dyn dyn;
510     FILE_ELF_SIZE_T dsz;
511     FILE_ELF_OFF_T doff;
512     uint64_t dynn;
513     int dynm;

515     doff = shdr->sh_offset;
516     dsz = sizeof (Elf_Dyn);

517     if (shdr->sh_size == 0 || shdr->sh_entsize == 0) {
518         (void) fprintf(stderr, ELF_ERR_DYNAMIC1,
519             File, EI->file);
520         return (ELF_READ_FAIL);
521     }

523     dynn = (shdr->sh_size / shdr->sh_entsize);
524     for (j = 0; j < dynn; j++) {
525         if (pread64(EI->elffd, &dyn, dsz, doff)
526             != dsz) ||
527             file_xlatetom(ELF_T_DYN, (char *)&dyn)
528             == 0) {
529             (void) fprintf(stderr, ELF_ERR_DYNAMIC2,
530                 File, EI->file);
531             return (ELF_READ_FAIL);

```

```

532     }
533
534     doff += dsize;
535
536     if ((dyn.d_tag == DT_SUNW_KMOD) &&
537         (dyn.d_un.d_val == 1)) {
538         EI->kmod = B_TRUE;
539     }
540 }
541
542
543 /*
544  * Definition time:
545  * - "not stripped" means that an executable file
546  *   contains a Symbol Table (.symtab)
547  * - "stripped" means that an executable file
548  *   does not contain a Symbol Table.
549  * When strip -l or strip -x is run, it strips the
550  * debugging information (.line section name (strip -l),
551  * .line, .debug*, .stabs*, .dwarf* section names
552  * and SHT_SUNW_DEBUGSTR and SHT_SUNW_DEBUG
553  * section types (strip -x), however the Symbol
554  * Table will still be present.
555  * Therefore, if
556  * - No Symbol Table present, then report
557  *   "stripped"
558  * - Symbol Table present with debugging
559  *   information (line number or debug section names,
560  *   or SHT_SUNW_DEBUGSTR or SHT_SUNW_DEBUG section
561  *   types) then report:
562  *   "not stripped"
563  * - Symbol Table present with no debugging
564  *   information (line number or debug section names,
565  *   or SHT_SUNW_DEBUGSTR or SHT_SUNW_DEBUG section
566  *   types) then report:
567  *   "not stripped, no debugging information
568  *   available"
569  */
570 if ((EI->stripped & E_NOSTRIP) == E_NOSTRIP)
571     continue;
572
573 if (!(EI->stripped & E_SYMTAB) &&
574     (shdr->sh_type == SHT_SYMTAB)) {
575     EI->stripped |= E_SYMTAB;
576     continue;
577 }
578
579 if (shdr->sh_name >= strtab_sz)
580     shnam = NULL;
581 else
582     shnam = &strtab[shdr->sh_name];
583
584 if (!(EI->stripped & E_DBGINF) &&
585     ((shdr->sh_type == SHT_SUNW_DEBUG) ||
586     (shdr->sh_type == SHT_SUNW_DEBUGSTR) ||
587     (shnam != NULL && is_in_list(shnam)))) {
588     EI->stripped |= E_DBGINF;
589 }
590 }
591 free(strtab);
592
593 return (ELF_READ_OKAY);
594 }

```

unchanged portion omitted

```

*****
44712 Fri Mar 1 17:10:00 2019
new/usr/src/cmd/file/file.c
10476 file(1) could be smatch clean
*****
_____unchanged_portion_omitted_____

710 /*
711 * def_context_tests() - default context-sensitive tests.
712 *   These are the last tests to be applied.
713 *   If no match is found, prints out "data".
714 */

716 static void
717 def_context_tests(void)
718 {
719     int    j;
720     int    nl;
721     char   ch;
722     int    len;

724     if (ccom() == 0)
725         goto notc;
726     while (fbuf[i] == '#') {
727         j = i;
728         while (fbuf[i++] != '\n') {
729             if (i - j > 255) {
730                 (void) printf(gettext("data\n"));
731                 return;
732             }
733             if (i >= fbsz)
734                 goto notc;
735         }
736         if (ccom() == 0)
737             goto notc;
738     }
739     check:
740     if (lookup(c) == 1) {
741         while ((ch = fbuf[i]) != ';' && ch != '{') {
742             if ((len = mblen(&fbuf[i], MB_CUR_MAX)) <= 0)
743                 len = 1;
744             i += len;
745             if (i >= fbsz)
746                 goto notc;
747         }
748         (void) printf(gettext("c program text"));
749         goto outa;
750     }
751     nl = 0;
752     while (fbuf[i] != '(') {
753         if (fbuf[i] <= 0)
754             goto notas;
755         if (fbuf[i] == ';') {
756             i++;
757             goto check;
758         }
759         if (fbuf[i++] == '\n')
760             if (nl++ > 6)
761                 goto notc;
762         if (i >= fbsz)
763             goto notc;
764     }
765     while (fbuf[i] != ')') {
766         if (fbuf[i++] == '\n')
767             if (nl++ > 6)
768                 goto notc;

```

```

769         if (i >= fbsz)
770             goto notc;
771     }
772     while (fbuf[i] != '{') {
773         if ((len = mblen(&fbuf[i], MB_CUR_MAX)) <= 0)
774             len = 1;
775         if (fbuf[i] == '\n')
776             if (nl++ > 6)
777                 goto notc;
778         i += len;
779         if (i >= fbsz)
780             goto notc;
781     }
782     (void) printf(gettext("c program text"));
783     goto outa;
784 notc:
785     i = 0;
786     /* reset to beginning of file again */
787     while (fbuf[i] == 'c' || fbuf[i] == 'C' || fbuf[i] == '!' ||
788           fbuf[i] == '*' || fbuf[i] == '\n') {
789         while (fbuf[i++] != '\n')
790             if (i >= fbsz)
791                 goto notfort;
792     }
793     if (lookup(fort) == 1) {
794         (void) printf(gettext("fortran program text"));
795         goto outa;
796     }
797 notfort:
798     /* looking for assembler program */
799     i = 0;
800     /* reset to beginning of file again */
801     /* assembler programs may contain */
802     /* c-style comments */
803     if (ccom() == 0)
804         goto notas;
805     if (ascom() == 0)
806         goto notas;
807     j = i - 1;
808     if (fbuf[i] == '.') {
809         i++;
810         if (lookup(as) == 1) {
811             (void) printf(gettext("assembler program text"));
812             goto outa;
813         } else if (j != -1 && fbuf[j] == '\n' && isalpha(fbuf[j + 2])) {
814             (void) printf(
815                 gettext("[nt]roff, tbl, or eqn input text"));
816             goto outa;
817         }
818     }
819     while (lookup(asc) == 0) {
820         if (ccom() == 0)
821             goto notas;
822         if (ascom() == 0)
823             goto notas;
824         while (fbuf[i] != '\n' && fbuf[i++] != ':') {
825             if (i >= fbsz)
826                 goto notas;
827         }
828         while (fbuf[i] == '\n' || fbuf[i] == ' ' || fbuf[i] == '\t')
829             if (i++ >= fbsz)
830                 goto notas;
831         j = i - 1;
832         if (fbuf[i] == '.') {
833             i++;
834             if (lookup(as) == 1) {
835                 (void) printf(
836                     gettext("assembler program text"));
837                 goto outa;
838             } else if (fbuf[j] == '\n' && isalpha(fbuf[j+2])) {

```

```

835         (void) printf(
836             gettext("[nt]roff, tbl, or eqn input "
837                 "text"));
838         goto outa;
839     }
840 }
841 }
842 (void) printf(gettext("assembler program text"));
843 goto outa;
844 notas:
845 /* start modification for multibyte env */
846 IS_ascii = 1;
847 if (fbsz < FBSZ)
848     Max = fbsz;
849 else
850     Max = FBSZ - MB_LEN_MAX; /* prevent cut of wchar read */
851 /* end modification for multibyte env */
852
853 for (i = 0; i < Max; /* null */)
854     if (fbuf[i] & 0200) {
855         IS_ascii = 0;
856         if ((fbuf[0] == '\100') &&
857             ((uchar_t)fbuf[1] == (uchar_t)\357)) {
858             if (fbuf[0] == '\100' && fbuf[1] == '\357') {
859                 (void) printf(gettext("troff output\n"));
860                 return;
861             }
862             /* start modification for multibyte env */
863             if ((length = mbtowc(&wchar, &fbuf[i], MB_CUR_MAX))
864                 <= 0 || !iswprint(wchar)) {
865                 (void) printf(gettext("data\n"));
866                 return;
867             }
868             i += length;
869         }
870         else
871             i++;
872     }
873 i = fbsz;
874 /* end modification for multibyte env */
875 if (mbuf.st_mode & (S_IXUSR|S_IXGRP|S_IXOTH))
876     (void) printf(gettext("commands text"));
877 else if (troffint(fbuf, fbsz))
878     (void) printf(gettext("troff intermediate output text"));
879 else if (english(fbuf, fbsz))
880     (void) printf(gettext("English text"));
881 else if (IS_ascii)
882     (void) printf(gettext("ascii text"));
883 else
884     (void) printf(gettext("text")); /* for multibyte env */
885 outa:
886 /*
887  * This code is to make sure that no MB char is cut in half
888  * while still being used.
889  */
890 fbsz = (fbsz < FBSZ ? fbsz : fbsz - MB_CUR_MAX + 1);
891 while (i < fbsz) {
892     if (isascii(fbuf[i])) {
893         i++;
894         continue;
895     } else {
896         if ((length = mbtowc(&wchar, &fbuf[i], MB_CUR_MAX))
897             <= 0 || !iswprint(wchar)) {
898             (void) printf(gettext(" with garbage\n"));
899             return;
900         }
901         i = i + length;

```

```

900     }
901     }
902     (void) printf("\n");
903 }

```

unchanged_portion_omitted

```

*****
22385 Fri Mar 1 17:10:00 2019
new/usr/src/cmd/file/magicutils.c
10476 file(1) could be smatch clean
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2007 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
27 /*      All Rights Reserved */

29 /*      Copyright (c) 1987, 1988 Microsoft Corporation */
30 /*      All Rights Reserved */

32 #pragma ident      "%Z%M% %I%      %E% SMI"

32 #include <stdio.h>
33 #include <stdlib.h>
34 #include <string.h>
35 #include <ctype.h>
36 #include <errno.h>
37 #include <limits.h>
38 #include <inttypes.h>
39 #include <sys/types.h>
40 #include <libintl.h>

42 /*
43  *      Types
44  */

46 #define BYTE      1
47 #define SHORT      2
48 #define LONG      4
49 #define LLONG      8
50 #define UBYTE      16
51 #define USHORT      32
52 #define ULONG      64
53 #define ULLONG      128
54 #define STR      256

56 /*
57  *      Opcodes
58  */

```

```

60 #define EQ      0
61 #define GT      1
62 #define LT      2
63 #define STRC      3      /* string compare */
64 #define ANY      4
65 #define AND      5
66 #define NSET      6      /* True if bit is not set */
67 #define SUB      64      /* or'ed in, SUBstitution string, for example */
68
69 /* %ld, %s, %lo mask: with bit 6 on, used to locate */
70 /* print formats */
71 *      Misc
72 */

74 #define BSZ      128
75 #define NENT      200

77 /*
78  *      Structure of magic file entry
79  */

81 struct      entry      {
82      char      e_level;      /* 0 or 1 */
83      off_t      e_off;      /* in bytes */
84      uint32_t      e_type;      /* BYTE, SHORT, STR, et al */
85      char      e_opcode;      /* EQ, GT, LT, ANY, AND, NSET */
86      uint64_t      e_mask;      /* if non-zero, mask value with this */
87      union      {
88          uint64_t      num;
89          char      *str;
90      }      e_value;
91      const char      *e_str;
92 };

unchanged_portion_omitted

190 /*
191 * f_mkmtab - fills mtab array of magic table entries with
192 * values from the file magfile.
193 * May be called more than once if multiple magic
194 * files were specified.
195 * Stores entries sequentially in one of two magic
196 * tables: mtab1, if first = 1; mtab2 otherwise.
197 *
198 * If -c option is specified, cflg is non-zero, and
199 * f_mkmtab() reports on errors in the magic file.
200 *
201 * Two magic tables may need to be created. The first
202 * one (mtab1) contains magic entries to be checked before
203 * the programmatic default position-sensitive tests in
204 * def_position_tests().
205 * The second one (mtab2) should start with the default
206 * /etc/magic file entries and is to be checked after
207 * the programmatic default position-sensitive tests in
208 * def_position_tests(). The parameter "first" would
209 * be 1 for the former set of tables, 0 for the latter
210 * set of magic tables.
211 * No mtab2 should be created if file will not be
212 * applying default tests; in that case, all magic table
213 * entries should be in mtab1.
214 *
215 * f_mkmtab returns 0 on success, -1 on error. The calling
216 * program is not expected to proceed after f_mkmtab()
217 * returns an error.
218 */

220 int

```

```

221 f_mkmtab(char *magfile, int cflg, int first)
222 {
223     Entry *mtab; /* generic magic table pointer */
224     Entry *ep; /* current magic table entry */
225     Entry *mend; /* one past last-allocated entry of mtab */
226     FILE *fp;
227     int lcnt = 0;
228     char buf[BSZ];
229     size_t tbsize;
230     size_t oldsize;
231
232     if (first) {
233         mtab = mtab1;
234         mend = mend1;
235         ep = epl;
236     } else {
237         mtab = mtab2;
238         mend = mend2;
239         ep = ep2;
240     }
241
242     /* mtab may have been allocated on a previous f_mkmtab call */
243     if (mtab == (Entry *)NULL) {
244         if ((mtab = calloc(NENT, sizeof (Entry))) == NULL) {
245             if ((mtab = calloc(sizeof (Entry), NENT)) == NULL) {
246                 int err = errno;
247                 (void) fprintf(stderr, gettext("%s: malloc "
248                 "failed: %s\n"), File, strerror(err));
249                 return (-1);
250             }
251             ep = mtab;
252             mend = &mtab[NENT];
253         }
254
255         errno = 0;
256         if ((fp = fopen(magfile, "r")) == NULL) {
257             int err = errno;
258             (void) fprintf(stderr, gettext("%s: %s: cannot open magic "
259             "file: %s\n"), File, magfile, err ? strerror(err) : "");
260             return (-1);
261         }
262         while (fgets(buf, BSZ, fp) != NULL) {
263             char *p = buf;
264             char *p2;
265             char *p3;
266             char opc;
267
268             /*
269             * ensure we have one extra entry allocated
270             * to mark end of the table, after the while loop
271             */
272             if (ep >= (mend - 1)) {
273                 oldsize = mend - mtab;
274                 tbsize = (NENT + oldsize) * sizeof (Entry);
275                 if ((mtab = realloc(mtab, tbsize)) == NULL) {
276                     int err = errno;
277                     (void) fprintf(stderr, gettext("%s: malloc "
278                     "failed: %s\n"), File, strerror(err));
279                     return (-1);
280                 } else {
281                     (void) memset(mtab + oldsize, 0,
282                     sizeof (Entry) * NENT);
283                     mend = &mtab[tbsize / sizeof (Entry)];
284                     ep = &mtab[oldsize-1];
285                 }

```

```

286     }
287
288     lcnt++;
289     if (*p == '\n' || *p == '#')
290         continue;
291
292     /* LEVEL */
293     if (*p == '>') {
294         ep->e_level = 1;
295         p++;
296     }
297     /* OFFSET */
298     p2 = strchr(p, '\t');
299     if (p2 == NULL) {
300         if (cflg)
301             (void) fprintf(stderr, gettext("%s: %s: format "
302             "error, no tab after %s on line %d\n"),
303             File, magfile, p, lcnt);
304         continue;
305     }
306     *p2++ = NULL;
307     ep->e_off = strtoul((const char *)p, (char **)NULL, 0);
308     while (*p2 == '\t')
309         p2++;
310     /* TYPE */
311     p = p2;
312     p2 = strchr(p, '\t');
313     if (p2 == NULL) {
314         if (cflg)
315             (void) fprintf(stderr, gettext("%s: %s: format "
316             "error, no tab after %s on line %d\n"),
317             File, magfile, p, lcnt);
318         continue;
319     }
320     *p2++ = NULL;
321     p3 = strchr(p, '&');
322     if (p3 != NULL) {
323         *p3++ = '\0';
324         ep->e_mask = strtoul((const char *)p3, (char **)NULL,
325         0); /* returns 0 or ULLONG_MAX on error */
326     } else {
327         ep->e_mask = 0ULL;
328     }
329     switch (*p) {
330     case 'd':
331         if (*(p+1) == NULL) {
332             /* d */
333             ep->e_type = LONG;
334         } else if (*(p+2) == NULL) { /* d? */
335             switch (*(p+1)) {
336             case 'C':
337                 case 'l':
338                     /* dC, dL */
339                     ep->e_type = BYTE;
340                     break;
341             case 'S':
342                 case '2':
343                     /* dS, d2 */
344                     ep->e_type = SHORT;
345                     break;
346             case 'I':
347                 case 'L':
348                 case '4':
349                     /* dI, dL, d4 */
350                     ep->e_type = LONG;
351

```

```

352             break;
353         case '8':
354             /* d8 */
355             ep->e_type = LLONG;
356             break;
357         default:
358             ep->e_type = LONG;
359             break;
360     }
361     }
362     break;
363 case 'l':
364     if (*(p+1) == 'l') { /* llong */
365         ep->e_type = LLONG;
366     } else { /* long */
367         ep->e_type = LONG;
368     }
369     break;
370 case 's':
371     if (*(p+1) == 'h') {
372         /* short */
373         ep->e_type = SHORT;
374     } else {
375         /* s or string */
376         ep->e_type = STR;
377     }
378     break;
379 case 'u':
380     if (*(p+1) == NULL) {
381         /* u */
382         ep->e_type = ULONG;
383     } else if (*(p+2) == NULL) { /* u? */
384         switch (*(p+1)) {
385             case 'C':
386             case 'l':
387                 /* uC, u1 */
388                 ep->e_type = UBYTE;
389                 break;
390             case 'S':
391             case '2':
392                 /* uS, u2 */
393                 ep->e_type = USHORT;
394                 break;
395             case 'I':
396             case 'L':
397             case '4':
398                 /* uI, uL, u4 */
399                 ep->e_type = ULONG;
400                 break;
401             case '8':
402                 /* u8 */
403                 ep->e_type = ULLONG;
404                 break;
405             default:
406                 ep->e_type = ULONG;
407                 break;
408         }
409     } else { /* u?* */
410         switch (*(p+1)) {
411             case 'b': /* ubyte */
412                 ep->e_type = UBYTE;
413                 break;
414             case 's': /* ushort */
415                 ep->e_type = USHORT;
416                 break;
417             case 'l':

```

```

418             if (*(p+2) == 'l') {
419                 /* ullong */
420                 ep->e_type = ULLONG;
421             } else {
422                 /* ulong */
423                 ep->e_type = ULONG;
424             }
425             break;
426         default:
427             /* default, same as "u" */
428             ep->e_type = ULONG;
429             break;
430     }
431     }
432     }
433     default:
434         /* retain (undocumented) default type */
435         ep->e_type = BYTE;
436         break;
437     }
438     if (ep->e_type == 0) {
439         ep->e_type = BYTE; /* default */
440     }
441     while (*p2 == '\t')
442         p2++;
443     /* OP-VALUE */
444     p = p2;
445     p2 = strchr(p, '\t');
446     if (p2 == NULL) {
447         if (cflg)
448             (void) fprintf(stderr, gettext("%s: %s: format "
449                 "error, no tab after %s on line %d\n"),
450                 File, magfile, p, lcnt);
451         continue;
452     }
453     *p2++ = NULL;
454     if (ep->e_type != STR) {
455         opc = *p++;
456         switch (opc) {
457             case '=':
458                 ep->e_opcode = EQ;
459                 break;
460             case '>':
461                 ep->e_opcode = GT;
462                 break;
463             case '<':
464                 ep->e_opcode = LT;
465                 break;
466             case 'x':
467                 ep->e_opcode = ANY;
468                 break;
469             case '&':
470                 ep->e_opcode = AND;
471                 break;
472             case '^':
473                 ep->e_opcode = NSET;
474                 break;
475             default:
476                 /* EQ (i.e. 0) is default */
477                 p--; /* since global ep->e_opcode=0 */
478         }
479     }
480     }
481     }
482     }
483     }

```



```
484         if (ep->e_opcode != ANY) {
485             if (ep->e_type != STR) {
486                 ep->e_value.num = strtoull((const char *)p,
487                     (char **)NULL, 0);
488             } else if ((ep->e_value.str =
489                 getstr(p, magfile)) == NULL) {
490                 return (-1);
491             }
492         }
493         p2 += strspn(p2, "\t");
494         /* STRING */
495         if ((ep->e_str = strdup(p2)) == NULL) {
496             int err = errno;
497             (void) fprintf(stderr, gettext("%s: malloc "
498                 "failed: %s\n"), File, strerror(err));
499             return (-1);
500         } else {
501             if ((p = strchr(ep->e_str, '\n')) != NULL)
502                 *p = '\0';
503             if (strchr(ep->e_str, '%') != NULL)
504                 ep->e_opcode |= SUB;
505         }
506         ep++;
507     }
508     /* end while (fgets) */
509     ep->e_off = -1L;      /* mark end of table */
510     if (first) {
511         mtab1 = mtab;
512         mend1 = mend;
513         ep1 = ep;
514     } else {
515         mtab2 = mtab;
516         mend2 = mend;
517         ep2 = ep;
518     }
519     if (fclose(fp) != 0) {
520         int err = errno;
521         (void) fprintf(stderr, gettext("%s: fclose failed: %s\n"),
522             File, strerror(err));
523         return (-1);
524     }
525     return (0);
526 }
```

unchanged_portion_omitted

```

*****
67503 Fri Mar 1 17:10:01 2019
new/usr/src/cmd/sgs/libld/common/args.c
code review from Robert
*****
_____unchanged_portion_omitted_____

1005 static int      opttitle = 0;
1006 /*
1007  * Parsing options pass1 for process_flags().
1008  */
1009 static uintptr_t
1010 parseopt_pass1(Of1_desc *ofl, int argc, char **argv, int *usage)
1011 {
1012     int      c, ndx = optind;

1014     /*
1015     * The -32, -64 and -ztarget options are special, in that we validate
1016     * them, but otherwise ignore them. libld.so (this code) is called
1017     * from the ld front end program. ld has already examined the
1018     * arguments to determine the output class and machine type of the
1019     * output object, as reflected in the version (32/64) of ld_main()
1020     * that was called and the value of the 'mach' argument passed.
1021     * By time execution reaches this point, these options have already
1022     * been seen and acted on.
1023     */
1024     while ((c = ld_getopt(ofl->ofl_lml, ndx, argc, argv)) != -1) {

1026         switch (c) {
1027         case '3':
1028             DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));

1030             /*
1031             * -32 is processed by ld to determine the output class.
1032             * Here we sanity check the option incase some other
1033             * -3* option is mistakenly passed to us.
1034             */
1035             if (optarg[0] != '2')
1036                 ld_eprintf(ofl, ERR_FATAL,
1037                     MSG_INTL(MSG_ARG_ILLEGAL),
1038                     MSG_ORIG(MSG_ARG_3), optarg);
1039             continue;

1041         case '6':
1042             DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));

1044             /*
1045             * -64 is processed by ld to determine the output class.
1046             * Here we sanity check the option incase some other
1047             * -6* option is mistakenly passed to us.
1048             */
1049             if (optarg[0] != '4')
1050                 ld_eprintf(ofl, ERR_FATAL,
1051                     MSG_INTL(MSG_ARG_ILLEGAL),
1052                     MSG_ORIG(MSG_ARG_6), optarg);
1053             continue;

1055         case 'a':
1056             DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, NULL));
1057             aflag = TRUE;
1058             break;

1060         case 'b':
1061             DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, NULL));
1062             bflag = TRUE;

```

```

1064         /*
1065         * This is a hack, and may be undone later.
1066         * The -b option is only used to build the Unix
1067         * kernel and its related kernel-mode modules.
1068         * We do not want those files to get a .SUNW_ldynsym
1069         * section. At least for now, the kernel makes no
1070         * use of .SUNW_ldynsym, and we do not want to use
1071         * the space to hold it. Therefore, we overload
1072         * the use of -b to also imply -znoldynsym.
1073         */
1074         ofl->ofl_flags |= FLG_OF_NOLDYNSYM;
1075         break;

1077     case 'c':
1078         DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));
1079         if (ofl->ofl_config)
1080             ld_eprintf(ofl, ERR_WARNING_NF,
1081                 MSG_INTL(MSG_ARG_MTONCE),
1082                 MSG_ORIG(MSG_ARG_C));
1083         else
1084             ofl->ofl_config = optarg;
1085         break;

1087     case 'C':
1088         DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, NULL));
1089         demangle_flag = 1;
1090         break;

1092     case 'd':
1093         DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));
1094         if ((optarg[0] == 'n') && (optarg[1] == '\0')) {
1095             if (dflag != SET_UNKNOWN)
1096                 ld_eprintf(ofl, ERR_WARNING_NF,
1097                     MSG_INTL(MSG_ARG_MTONCE),
1098                     MSG_ORIG(MSG_ARG_D));
1099             else
1100                 dflag = SET_FALSE;
1101         } else if ((optarg[0] == 'y') && (optarg[1] == '\0')) {
1102             if (dflag != SET_UNKNOWN)
1103                 ld_eprintf(ofl, ERR_WARNING_NF,
1104                     MSG_INTL(MSG_ARG_MTONCE),
1105                     MSG_ORIG(MSG_ARG_D));
1106             else
1107                 dflag = SET_TRUE;
1108         } else {
1109             ld_eprintf(ofl, ERR_FATAL,
1110                 MSG_INTL(MSG_ARG_ILLEGAL),
1111                 MSG_ORIG(MSG_ARG_D), optarg);
1112         }
1113         break;

1115     case 'e':
1116         DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));
1117         if (ofl->ofl_entry)
1118             ld_eprintf(ofl, ERR_WARNING_NF,
1119                 MSG_INTL(MSG_MARG_MTONCE),
1120                 MSG_INTL(MSG_MARG_ENTRY));
1121         else
1122             ofl->ofl_entry = (void *)optarg;
1123         break;

1125     case 'f':
1126         DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));
1127         if (ofl->ofl_filtrees &&
1128             (!(ofl->ofl_flags & FLG_OF_AUX))) {
1129             ld_eprintf(ofl, ERR_FATAL,

```

```

1130         MSG_INTL(MSG_MARG_INCOMP),
1131         MSG_INTL(MSG_MARG_FILTER_AUX),
1132         MSG_INTL(MSG_MARG_FILTER));
1133     } else {
1134         if ((ofl->ofl_filtees =
1135             add_string(ofl->ofl_filtees, optarg)) ==
1136             (const char *)S_ERROR)
1137             return (S_ERROR);
1138         ofl->ofl_flags |= FLG_OF_AUX;
1139     }
1140     break;

1142 case 'F':
1143     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
1144     if (ofl->ofl_filtees &&
1145         (ofl->ofl_flags & FLG_OF_AUX)) {
1146         ld_eprintf(ofl, ERR_FATAL,
1147                 MSG_INTL(MSG_MARG_INCOMP),
1148                 MSG_INTL(MSG_MARG_FILTER),
1149                 MSG_INTL(MSG_MARG_FILTER_AUX));
1150     } else {
1151         if ((ofl->ofl_filtees =
1152             add_string(ofl->ofl_filtees, optarg)) ==
1153             (const char *)S_ERROR)
1154             return (S_ERROR);
1155     }
1156     break;

1158 case 'h':
1159     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
1160     if (ofl->ofl_soname)
1161         ld_eprintf(ofl, ERR_WARNING_NF,
1162                 MSG_INTL(MSG_MARG_MTONCE),
1163                 MSG_INTL(MSG_MARG_SONAME));
1164     else
1165         ofl->ofl_soname = (const char *)optarg;
1166     break;

1168 case 'i':
1169     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
1170     ofl->ofl_flags |= FLG_OF_IGNENV;
1171     break;

1173 case 'I':
1174     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
1175     if (ofl->ofl_interp)
1176         ld_eprintf(ofl, ERR_WARNING_NF,
1177                 MSG_INTL(MSG_ARG_MTONCE),
1178                 MSG_ORIG(MSG_ARG_CI));
1179     else
1180         ofl->ofl_interp = (const char *)optarg;
1181     break;

1183 case 'l':
1184     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
1185     /*
1186     * For now, count any library as a shared object. This
1187     * is used to size the internal symbol cache. This
1188     * value is recalculated later on actual file processing
1189     * to get an accurate shared object count.
1190     */
1191     ofl->ofl_soscnt++;
1192     break;

1194 case 'm':
1195     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));

```

```

1196         ofl->ofl_flags |= FLG_OF_GENMAP;
1197     break;

1199 case 'o':
1200     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
1201     if (ofl->ofl_name)
1202         ld_eprintf(ofl, ERR_WARNING_NF,
1203                 MSG_INTL(MSG_MARG_MTONCE),
1204                 MSG_INTL(MSG_MARG_OUTFILE));
1205     else
1206         ofl->ofl_name = (const char *)optarg;
1207     break;

1209 case 'p':
1210     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));

1212     /*
1213     * Multiple instances of this option may occur. Each
1214     * additional instance is effectively concatenated to
1215     * the previous separated by a colon.
1216     */
1217     if (*optarg != '\0') {
1218         if ((ofl->ofl_audit =
1219             add_string(ofl->ofl_audit,
1220                 optarg)) == (const char *)S_ERROR)
1221             return (S_ERROR);
1222     }
1223     break;

1225 case 'P':
1226     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));

1228     /*
1229     * Multiple instances of this option may occur. Each
1230     * additional instance is effectively concatenated to
1231     * the previous separated by a colon.
1232     */
1233     if (*optarg != '\0') {
1234         if ((ofl->ofl_depaudit =
1235             add_string(ofl->ofl_depaudit,
1236                 optarg)) == (const char *)S_ERROR)
1237             return (S_ERROR);
1238     }
1239     break;

1241 case 'r':
1242     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
1243     otype = OT_RELOC;
1244     break;

1246 case 'R':
1247     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));

1249     /*
1250     * Multiple instances of this option may occur. Each
1251     * additional instance is effectively concatenated to
1252     * the previous separated by a colon.
1253     */
1254     if (*optarg != '\0') {
1255         if ((ofl->ofl_rpath =
1256             add_string(ofl->ofl_rpath,
1257                 optarg)) == (const char *)S_ERROR)
1258             return (S_ERROR);
1259     }
1260     break;

```

```

1262     case 's':
1263         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
1264         sflag = TRUE;
1265         break;
1267     case 't':
1268         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
1269         ofl->ofl_flags |= FLG_OF_NOWARN;
1270         break;
1272     case 'u':
1273         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
1274         break;
1276     case 'z':
1277         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
1279         /*
1280          * Skip comma that might be present between -z and its
1281          * argument (e.g. if -Wl,-z,assert-deflib was passed).
1282          */
1283         if (strncmp(optarg, MSG_ORIG(MSG_STR_COMMA),
1284             MSG_STR_COMMA_SIZE) == 0)
1285             optarg++;
1287         /*
1288          * For specific help, print our usage message and exit
1289          * immediately to ensure a 0 return code.
1290          */
1291         if (strncmp(optarg, MSG_ORIG(MSG_ARG_HELP),
1292             MSG_ARG_HELP_SIZE) == 0) {
1293             usage_mesg(TRUE);
1294             exit(0);
1295         }
1297         /*
1298          * For some options set a flag - further consistency
1299          * checks will be carried out in check_flags().
1300          */
1301         if ((strncmp(optarg, MSG_ORIG(MSG_ARG_LD32),
1302             MSG_ARG_LD32_SIZE) == 0) ||
1303             (strncmp(optarg, MSG_ORIG(MSG_ARG_LD64),
1304             MSG_ARG_LD64_SIZE) == 0)) {
1305             if (createargv(ofl, usage) == S_ERROR)
1306                 return (S_ERROR);
1308         } else if (
1309             strcmp(optarg, MSG_ORIG(MSG_ARG_DEFS)) == 0) {
1310             if (zdflag != SET_UNKNOWN)
1311                 ld_eprintf(ofl, ERR_WARNING_NF,
1312                     MSG_INTL(MSG_ARG_MTONCE),
1313                     MSG_ORIG(MSG_ARG_ZDEFNODEF));
1314             else
1315                 zdflag = SET_TRUE;
1316             ofl->ofl_guideflags |= FLG_OFG_NO_DEFS;
1317         } else if (strcmp(optarg,
1318             MSG_ORIG(MSG_ARG_NODEFS)) == 0) {
1319             if (zdflag != SET_UNKNOWN)
1320                 ld_eprintf(ofl, ERR_WARNING_NF,
1321                     MSG_INTL(MSG_ARG_MTONCE),
1322                     MSG_ORIG(MSG_ARG_ZDEFNODEF));
1323             else
1324                 zdflag = SET_FALSE;
1325             ofl->ofl_guideflags |= FLG_OFG_NO_DEFS;
1326         } else if (strcmp(optarg,
1327             MSG_ORIG(MSG_ARG_TEXT)) == 0) {

```

```

1328         if (ztflag &&
1329             (ztflag != MSG_ORIG(MSG_ARG_ZTEXT)))
1330             ld_eprintf(ofl, ERR_FATAL,
1331                 MSG_INTL(MSG_ARG_INCOMP),
1332                 MSG_ORIG(MSG_ARG_ZTEXT),
1333                 ztflag);
1334         ztflag = MSG_ORIG(MSG_ARG_ZTEXT);
1335     } else if (strcmp(optarg,
1336         MSG_ORIG(MSG_ARG_TEXTOFF)) == 0) {
1337         if (ztflag &&
1338             (ztflag != MSG_ORIG(MSG_ARG_ZTEXTOFF)))
1339             ld_eprintf(ofl, ERR_FATAL,
1340                 MSG_INTL(MSG_ARG_INCOMP),
1341                 MSG_ORIG(MSG_ARG_ZTEXTOFF),
1342                 ztflag);
1343         ztflag = MSG_ORIG(MSG_ARG_ZTEXTOFF);
1344     } else if (strcmp(optarg,
1345         MSG_ORIG(MSG_ARG_TEXTWARN)) == 0) {
1346         if (ztflag &&
1347             (ztflag != MSG_ORIG(MSG_ARG_ZTEXTWARN)))
1348             ld_eprintf(ofl, ERR_FATAL,
1349                 MSG_INTL(MSG_ARG_INCOMP),
1350                 MSG_ORIG(MSG_ARG_ZTEXTWARN),
1351                 ztflag);
1352         ztflag = MSG_ORIG(MSG_ARG_ZTEXTWARN);
1354     /*
1355      * For other options simply set the ofl flags directly.
1356      */
1357     } else if (strcmp(optarg,
1358         MSG_ORIG(MSG_ARG_RESCAN)) == 0) {
1359         ofl->ofl_flags1 |= FLG_OF1_RESCAN;
1360     } else if (strcmp(optarg,
1361         MSG_ORIG(MSG_ARG_ABSEXEC)) == 0) {
1362         ofl->ofl_flags1 |= FLG_OF1_ABSEXEC;
1363     } else if (strcmp(optarg,
1364         MSG_ORIG(MSG_ARG_LOADFLTR)) == 0) {
1365         zlflag = TRUE;
1366     } else if (strcmp(optarg,
1367         MSG_ORIG(MSG_ARG_NORELOC)) == 0) {
1368         ofl->ofl_dtflags_1 |= DF_1_NORELOC;
1369     } else if (strcmp(optarg,
1370         MSG_ORIG(MSG_ARG_NOVERSION)) == 0) {
1371         ofl->ofl_flags |= FLG_OF_NOVERSEC;
1372     } else if (strcmp(optarg,
1373         MSG_ORIG(MSG_ARG_MULDEFS)) == 0) {
1374         ofl->ofl_flags |= FLG_OF_MULDEFS;
1375     } else if (strcmp(optarg,
1376         MSG_ORIG(MSG_ARG_REDLOCSYM)) == 0) {
1377         ofl->ofl_flags |= FLG_OF_REDLSYM;
1378     } else if (strcmp(optarg,
1379         MSG_ORIG(MSG_ARG_INITFIRST)) == 0) {
1380         ofl->ofl_dtflags_1 |= DF_1_INITFIRST;
1381     } else if (strcmp(optarg,
1382         MSG_ORIG(MSG_ARG_NODELETE)) == 0) {
1383         ofl->ofl_dtflags_1 |= DF_1_NODELETE;
1384     } else if (strcmp(optarg,
1385         MSG_ORIG(MSG_ARG_NOPARTIAL)) == 0) {
1386         ofl->ofl_flags1 |= FLG_OF1_NOPARTI;
1387     } else if (strcmp(optarg,
1388         MSG_ORIG(MSG_ARG_NOOPEN)) == 0) {
1389         ofl->ofl_dtflags_1 |= DF_1_NOOPEN;
1390     } else if (strcmp(optarg,
1391         MSG_ORIG(MSG_ARG_NOW)) == 0) {
1392         ofl->ofl_dtflags_1 |= DF_1_NOW;
1393         ofl->ofl_dtflags |= DF_BIND_NOW;

```

```

1394     } else if (strcmp(optarg,
1395                MSG_ORIG(MSG_ARG_ORIGIN)) == 0) {
1396         ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
1397         ofl->ofl_dtflags |= DF_ORIGIN;
1398     } else if (strcmp(optarg,
1399                MSG_ORIG(MSG_ARG_NODEFAULTLIB)) == 0) {
1400         ofl->ofl_dtflags_1 |= DF_1_NODEFLIB;
1401     } else if (strcmp(optarg,
1402                MSG_ORIG(MSG_ARG_NODUMP)) == 0) {
1403         ofl->ofl_dtflags_1 |= DF_1_NODUMP;
1404     } else if (strcmp(optarg,
1405                MSG_ORIG(MSG_ARG_ENDFILTEE)) == 0) {
1406         ofl->ofl_dtflags_1 |= DF_1_ENDFILTEE;
1407     } else if (strcmp(optarg,
1408                MSG_ORIG(MSG_ARG_VERBOSE)) == 0) {
1409         ofl->ofl_flags |= FLG_OF_VERBOSE;
1410     } else if (strcmp(optarg,
1411                MSG_ORIG(MSG_ARG_COMBRELOC)) == 0) {
1412         ofl->ofl_flags |= FLG_OF_COMREL;
1413     } else if (strcmp(optarg,
1414                MSG_ORIG(MSG_ARG_NOCOMBRELOC)) == 0) {
1415         ofl->ofl_flags |= FLG_OF_NOCOMREL;
1416     } else if (strcmp(optarg,
1417                MSG_ORIG(MSG_ARG_NOCOMPSTRTAB)) == 0) {
1418         ofl->ofl_flags1 |= FLG_OF1_NCSTTAB;
1419     } else if (strcmp(optarg,
1420                MSG_ORIG(MSG_ARG_NOINTERP)) == 0) {
1421         ofl->ofl_flags1 |= FLG_OF1_NOINTRP;
1422     } else if (strcmp(optarg,
1423                MSG_ORIG(MSG_ARG_INTERPOSE)) == 0) {
1424         zinflag = TRUE;
1425     } else if (strcmp(optarg,
1426                MSG_ORIG(MSG_ARG_IGNORE)) == 0) {
1427         ofl->ofl_flags1 |= FLG_OF1_IGNPRC;
1428     } else if (strcmp(optarg,
1429                MSG_ORIG(MSG_ARG_RELAXRELOC)) == 0) {
1430         ofl->ofl_flags1 |= FLG_OF1_RLXREL;
1431     } else if (strcmp(optarg,
1432                MSG_ORIG(MSG_ARG_NORELAXRELOC)) == 0) {
1433         ofl->ofl_flags1 |= FLG_OF1_NRLXREL;
1434     } else if (strcmp(optarg,
1435                MSG_ORIG(MSG_ARG_NOLDYNSYM)) == 0) {
1436         ofl->ofl_flags |= FLG_OF_NOLDYNSYM;
1437     } else if (strcmp(optarg,
1438                MSG_ORIG(MSG_ARG_GLOBAUDIT)) == 0) {
1439         ofl->ofl_dtflags_1 |= DF_1_GLOBAUDIT;
1440     } else if (strcmp(optarg,
1441                MSG_ORIG(MSG_ARG_NOSIGHANDLER)) == 0) {
1442         ofl->ofl_flags1 |= FLG_OF1_NOSGHND;
1443     } else if (strcmp(optarg,
1444                MSG_ORIG(MSG_ARG_SYMBOLCAP)) == 0) {
1445         ofl->ofl_flags |= FLG_OF_OTOSCAP;
1446     }
1447     /*
1448     * Check archive group usage
1449     * -z rescan-start ... -z rescan-end
1450     * to ensure they don't overlap and are well formed.
1451     */
1452     } else if (strcmp(optarg,
1453                MSG_ORIG(MSG_ARG_RESCAN_START)) == 0) {
1454         if (ofl->ofl_ars_gsndx == 0) {
1455             ofl->ofl_ars_gsndx = ndx;
1456         } else if (ofl->ofl_ars_gsndx > 0) {
1457             /* Another group is still open */
1458             ld_eprintf(ofl, ERR_FATAL,
1459                MSG_INTL(MSG_ARG_AR_GRP_OLAP),

```

```

1460             MSG_INTL(MSG_MARG_AR_GRP));
1461             /* Don't report cascading errors */
1462             ofl->ofl_ars_gsndx = -1;
1463         }
1464     } else if (strcmp(optarg,
1465                MSG_ORIG(MSG_ARG_RESCAN_END)) == 0) {
1466         if (ofl->ofl_ars_gsndx > 0) {
1467             ofl->ofl_ars_gsndx = 0;
1468         } else if (ofl->ofl_ars_gsndx == 0) {
1469             /* There was no matching begin */
1470             ld_eprintf(ofl, ERR_FATAL,
1471                MSG_INTL(MSG_ARG_AR_GRP_BAD),
1472                MSG_INTL(MSG_MARG_AR_GRP_END),
1473                MSG_INTL(MSG_MARG_AR_GRP_START));
1474             /* Don't report cascading errors */
1475             ofl->ofl_ars_gsndx = -1;
1476         }
1477     }
1478     /*
1479     * If -z wrap is seen, enter the symbol to be wrapped
1480     * into the wrap AVL tree.
1481     */
1482     } else if (strcmp(optarg, MSG_ORIG(MSG_ARG_WRAP),
1483                MSG_ARG_WRAP_SIZE) == 0) {
1484         if (ld_wrap_enter(ofl,
1485                optarg + MSG_ARG_WRAP_SIZE) == NULL)
1486             return (S_ERROR);
1487     } else if (strcmp(optarg, MSG_ORIG(MSG_ARG_ASLR),
1488                MSG_ARG_ASLR_SIZE) == 0) {
1489         char *p = optarg + MSG_ARG_ASLR_SIZE;
1490         if (*p == '\0') {
1491             ofl->ofl_aslr = 1;
1492         } else if (*p == '=') {
1493             p++;
1494         }
1495         if ((strcmp(p,
1496                MSG_ORIG(MSG_ARG_ENABLED)) == 0) ||
1497             (strcmp(p,
1498                MSG_ORIG(MSG_ARG_ENABLE)) == 0)) {
1499             ofl->ofl_aslr = 1;
1500         } else if ((strcmp(p,
1501                MSG_ORIG(MSG_ARG_DISABLED)) == 0) ||
1502             (strcmp(p,
1503                MSG_ORIG(MSG_ARG_DISABLE)) == 0)) {
1504             ofl->ofl_aslr = -1;
1505         } else {
1506             ld_eprintf(ofl, ERR_FATAL,
1507                MSG_INTL(MSG_ARG_ILLEGAL),
1508                MSG_ORIG(MSG_ARG_ZASLR), p);
1509             return (S_ERROR);
1510         }
1511     } else {
1512         ld_eprintf(ofl, ERR_FATAL,
1513                MSG_INTL(MSG_ARG_ILLEGAL),
1514                MSG_ORIG(MSG_ARG_Z), optarg);
1515         return (S_ERROR);
1516     }
1517     } else if ((strcmp(optarg, MSG_ORIG(MSG_ARG_GUIDE),
1518                MSG_ARG_GUIDE_SIZE) == 0) &&
1519                ((optarg[MSG_ARG_GUIDE_SIZE] == '=') ||
1520                (optarg[MSG_ARG_GUIDE_SIZE] == '\0'))) {
1521         if (!guidance_parse(ofl, optarg))
1522             return (S_ERROR);
1523     } else if (strcmp(optarg,
1524                MSG_ORIG(MSG_ARG_FATWARN)) == 0) {
1525         if (zfwflag == SET_FALSE) {

```

```

1526         ld_eprintf(ofl, ERR_WARNING_NF,
1527                   MSG_INTL(MSG_ARG_MTONCE),
1528                   MSG_ORIG(MSG_ARG_ZFATWNOFATW));
1529     } else {
1530         zfwflag = SET_TRUE;
1531         ofl->ofl_flags |= FLG_OF_FATWARN;
1532     }
1533     } else if (strcmp(optarg,
1534                   MSG_ORIG(MSG_ARG_NOFATWARN)) == 0) {
1535         if (zfwflag == SET_TRUE)
1536             ld_eprintf(ofl, ERR_WARNING_NF,
1537                       MSG_INTL(MSG_ARG_MTONCE),
1538                       MSG_ORIG(MSG_ARG_ZFATWNOFATW));
1539         else
1540             zfwflag = SET_FALSE;
1541
1542     /*
1543     * Process everything related to -z assert-deflib. This
1544     * must be done in pass 1 because it gets used in pass
1545     * 2.
1546     */
1547     } else if (strcmp(optarg, MSG_ORIG(MSG_ARG_ASSDEFLIB),
1548                   MSG_ARG_ASSDEFLIB_SIZE) == 0) {
1549         if (assdeflib_parse(ofl, optarg) != TRUE)
1550             return (S_ERROR);
1551
1552     /*
1553     * Process new-style output type specification, which
1554     * we'll use in pass 2 and throughout.
1555     */
1556     #endif /* ! codereview */
1557     } else if (strcmp(optarg, MSG_ORIG(MSG_ARG_TYPE),
1558                   MSG_ARG_TYPE_SIZE) == 0) {
1559         char *p = optarg + MSG_ARG_TYPE_SIZE;
1560         if (*p != '=') {
1561             ld_eprintf(ofl, ERR_FATAL,
1562                       MSG_INTL(MSG_ARG_ILLEGAL),
1563                       MSG_ORIG(MSG_ARG_Z), optarg);
1564             return (S_ERROR);
1565         }
1566
1567         p++;
1568         if (strcmp(p,
1569                   MSG_ORIG(MSG_ARG_TYPE_RELOC)) == 0) {
1570             otype = OT_RELOC;
1571         } else if (strcmp(p,
1572                   MSG_ORIG(MSG_ARG_TYPE_EXEC)) == 0) {
1573             otype = OT_EXEC;
1574         } else if (strcmp(p,
1575                   MSG_ORIG(MSG_ARG_TYPE_SHARED)) == 0) {
1576             otype = OT_SHARED;
1577         } else if (strcmp(p,
1578                   MSG_ORIG(MSG_ARG_TYPE_KMOD)) == 0) {
1579             otype = OT_KMOD;
1580         } else {
1581             ld_eprintf(ofl, ERR_FATAL,
1582                       MSG_INTL(MSG_ARG_ILLEGAL),
1583                       MSG_ORIG(MSG_ARG_Z), optarg);
1584             return (S_ERROR);
1585         }
1586
1587     /*
1588     * The following options just need validation as they
1589     * are interpreted on the second pass through the
1590     * command line arguments.
1591     */
1592     } else if (

```

```

1592         strcmp(optarg, MSG_ORIG(MSG_ARG_INITARRAY),
1593               MSG_ARG_INITARRAY_SIZE) &&
1594         strcmp(optarg, MSG_ORIG(MSG_ARG_FINIARRAY),
1595               MSG_ARG_FINIARRAY_SIZE) &&
1596         strcmp(optarg, MSG_ORIG(MSG_ARG_PREINITARRAY),
1597               MSG_ARG_PREINITARRAY_SIZE) &&
1598         strcmp(optarg, MSG_ORIG(MSG_ARG_RTLDINFO),
1599               MSG_ARG_RTLDINFO_SIZE) &&
1600         strcmp(optarg, MSG_ORIG(MSG_ARG_DTRACE),
1601               MSG_ARG_DTRACE_SIZE) &&
1602         strcmp(optarg, MSG_ORIG(MSG_ARG_ALLEXTRT)) &&
1603         strcmp(optarg, MSG_ORIG(MSG_ARG_DFLEXTRT)) &&
1604         strcmp(optarg, MSG_ORIG(MSG_ARG_DIRECT)) &&
1605         strcmp(optarg, MSG_ORIG(MSG_ARG_NODIRECT)) &&
1606         strcmp(optarg, MSG_ORIG(MSG_ARG_GROUPEM)) &&
1607         strcmp(optarg, MSG_ORIG(MSG_ARG_LAZYLOAD)) &&
1608         strcmp(optarg, MSG_ORIG(MSG_ARG_NOGROUPEM)) &&
1609         strcmp(optarg, MSG_ORIG(MSG_ARG_NOLAZYLOAD)) &&
1610         strcmp(optarg, MSG_ORIG(MSG_ARG_NODEFERRED)) &&
1611         strcmp(optarg, MSG_ORIG(MSG_ARG_RECORD)) &&
1612         strcmp(optarg, MSG_ORIG(MSG_ARG_ALTEXEC64)) &&
1613         strcmp(optarg, MSG_ORIG(MSG_ARG_WEAKEXT)) &&
1614         strcmp(optarg, MSG_ORIG(MSG_ARG_TARGET),
1615               MSG_ARG_TARGET_SIZE) &&
1616         strcmp(optarg, MSG_ORIG(MSG_ARG_RESCAN_NOW)) &&
1617         strcmp(optarg, MSG_ORIG(MSG_ARG_DEFERRED)) {
1618             ld_eprintf(ofl, ERR_FATAL,
1619                       MSG_INTL(MSG_ARG_ILLEGAL),
1620                       MSG_ORIG(MSG_ARG_Z), optarg);
1621         }
1622     }
1623     break;
1624
1625     case 'D':
1626     /*
1627     * If we have not yet read any input files go ahead
1628     * and process any debugging options (this allows any
1629     * argument processing, entrance criteria and library
1630     * initialization to be displayed). Otherwise, if an
1631     * input file has been seen, skip interpretation until
1632     * process_files (this allows debugging to be turned
1633     * on and off around individual groups of files).
1634     */
1635     Dflag = 1;
1636     if (ofl->ofl_objscnt == 0) {
1637         if (dbg_setup(ofl, optarg, 2) == 0)
1638             return (S_ERROR);
1639     }
1640
1641     /*
1642     * A diagnostic can only be provided after dbg_setup().
1643     * As this is the first diagnostic that can be produced
1644     * by ld(1), issue a title for timing and basic output.
1645     */
1646     if ((optitle == 0) && DBG_ENABLED) {
1647         optitle++;
1648         DBG_CALL(Dbg_basic_options(ofl->ofl_lml));
1649     }
1650     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
1651     break;
1652
1653     case 'B':
1654     DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
1655     if (strcmp(optarg, MSG_ORIG(MSG_ARG_DIRECT)) == 0) {
1656         if (Bdflag == SET_FALSE) {
1657             ld_eprintf(ofl, ERR_FATAL,

```

```

1658         MSG_INTL(MSG_ARG_INCOMP),
1659         MSG_ORIG(MSG_ARG_BNODIRECT),
1660         MSG_ORIG(MSG_ARG_BDIRECT));
1661     } else {
1662         Bdflag = SET_TRUE;
1663         ofl->ofl_guideflags |= FLG_OFG_NO_DB;
1664     }
1665 } else if (strcmp(optarg,
1666             MSG_ORIG(MSG_ARG_NODIRECT)) == 0) {
1667     if (Bdflag == SET_TRUE) {
1668         ld_eprintf(ofl, ERR_FATAL,
1669                 MSG_INTL(MSG_ARG_INCOMP),
1670                 MSG_ORIG(MSG_ARG_BDIRECT),
1671                 MSG_ORIG(MSG_ARG_BNODIRECT));
1672     } else {
1673         Bdflag = SET_FALSE;
1674         ofl->ofl_guideflags |= FLG_OFG_NO_DB;
1675     }
1676 } else if (strcmp(optarg,
1677             MSG_ORIG(MSG_STR_SYMBOLIC)) == 0)
1678     Bsflag = TRUE;
1679 else if (strcmp(optarg, MSG_ORIG(MSG_ARG_REDUCE)) == 0)
1680     ofl->ofl_flags |= FLG_OF_PROCRED;
1681 else if (strcmp(optarg, MSG_ORIG(MSG_STR_LOCAL)) == 0)
1682     Biflag = TRUE;
1683 else if (strcmp(optarg, MSG_ORIG(MSG_ARG_GROUP)) == 0)
1684     Bgflag = TRUE;
1685 else if (strcmp(optarg,
1686             MSG_ORIG(MSG_STR_ELIMINATE)) == 0)
1687     Beflag = TRUE;
1688 else if (strcmp(optarg,
1689             MSG_ORIG(MSG_ARG_TRANSLATOR)) == 0) {
1690     ld_eprintf(ofl, ERR_WARNING,
1691             MSG_INTL(MSG_ARG_UNSUPPORTED),
1692             MSG_ORIG(MSG_ARG_BTRANSLATOR));
1693 } else if (strcmp(optarg,
1694             MSG_ORIG(MSG_STR_LD_DYNAMIC)) &&
1695             strcmp(optarg, MSG_ORIG(MSG_ARG_STATIC))) {
1696     ld_eprintf(ofl, ERR_FATAL,
1697             MSG_INTL(MSG_ARG_ILLEGAL),
1698             MSG_ORIG(MSG_ARG_CB), optarg);
1699 }
1700 break;

1702 case 'G':
1703     DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, NULL));
1704     otype = OT_SHARED;
1705     break;

1707 case 'L':
1708     DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));
1709     break;

1711 case 'M':
1712     DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));
1713     if (aplist_append(&(ofl->ofl_maps), optarg,
1714                     AL_CNT_OF_MAPFILES) == NULL)
1715         return (S_ERROR);
1716     break;

1718 case 'N':
1719     DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));
1720     break;

1722 case 'Q':
1723     DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));

```

```

1724     if ((optarg[0] == 'n') && (optarg[1] == '\0')) {
1725         if (Qflag != SET_UNKNOWN)
1726             ld_eprintf(ofl, ERR_WARNING_NF,
1727                     MSG_INTL(MSG_ARG_MTONCE),
1728                     MSG_ORIG(MSG_ARG_CQ));
1729         else
1730             Qflag = SET_FALSE;
1731     } else if ((optarg[0] == 'y') && (optarg[1] == '\0')) {
1732         if (Qflag != SET_UNKNOWN)
1733             ld_eprintf(ofl, ERR_WARNING_NF,
1734                     MSG_INTL(MSG_ARG_MTONCE),
1735                     MSG_ORIG(MSG_ARG_CQ));
1736         else
1737             Qflag = SET_TRUE;
1738     } else {
1739         ld_eprintf(ofl, ERR_FATAL,
1740                 MSG_INTL(MSG_ARG_ILLEGAL),
1741                 MSG_ORIG(MSG_ARG_CQ), optarg);
1742     }
1743     break;

1745 case 'S':
1746     DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));
1747     if (aplist_append(&lib_support, optarg,
1748                     AL_CNT_SUPPORT) == NULL)
1749         return (S_ERROR);
1750     break;

1752 case 'V':
1753     DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, NULL));
1754     if (!Vflag)
1755         (void) fprintf(stderr, MSG_ORIG(MSG_STR_STRNL),
1756                 ofl->ofl_sgsid);
1757     Vflag = TRUE;
1758     break;

1760 case 'Y':
1761     DBG_CALL(DBG_args_option(ofl->ofl_lml, ndx, c, optarg));
1762     if (strncmp(optarg, MSG_ORIG(MSG_ARG_LCOM), 2) == 0) {
1763         if (Llibdir)
1764             ld_eprintf(ofl, ERR_WARNING_NF,
1765                     MSG_INTL(MSG_ARG_MTONCE),
1766                     MSG_ORIG(MSG_ARG_CYL));
1767         else
1768             Llibdir = optarg + 2;
1769     } else if (strncmp(optarg,
1770                     MSG_ORIG(MSG_ARG_UCOM), 2) == 0) {
1771         if (Ulibdir)
1772             ld_eprintf(ofl, ERR_WARNING_NF,
1773                     MSG_INTL(MSG_ARG_MTONCE),
1774                     MSG_ORIG(MSG_ARG_CYU));
1775         else
1776             Ulibdir = optarg + 2;
1777     } else if (strncmp(optarg,
1778                     MSG_ORIG(MSG_ARG_PCOM), 2) == 0) {
1779         if (Plibpath)
1780             ld_eprintf(ofl, ERR_WARNING_NF,
1781                     MSG_INTL(MSG_ARG_MTONCE),
1782                     MSG_ORIG(MSG_ARG_CYP));
1783         else
1784             Plibpath = optarg + 2;
1785     } else {
1786         ld_eprintf(ofl, ERR_FATAL,
1787                 MSG_INTL(MSG_ARG_ILLEGAL),
1788                 MSG_ORIG(MSG_ARG_CY), optarg);
1789     }

```

```

1790         break;
1792     case '?':
1793         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
1794         /*
1795          * If the option character is '-', we're looking at a
1796          * long option which couldn't be translated, display a
1797          * more useful error.
1798          */
1799         if (optopt == '-') {
1800             eprintf(ofl->ofl_lml, ERR_FATAL,
1801                 MSG_INTL(MSG_ARG_LONG_UNKNOWN),
1802                 argv[optind-1]);
1803         } else {
1804             eprintf(ofl->ofl_lml, ERR_FATAL,
1805                 MSG_INTL(MSG_ARG_UNKNOWN), optopt);
1806         }
1807         (*usage)++;
1808         break;
1810     default:
1811         break;
1812 }
1814 /*
1815  * Update the argument index for the next getopt() iteration.
1816  */
1817     ndx = optind;
1818 }
1819     return (1);
1820 }
1822 /*
1823  * Parsing options pass2 for
1824  */
1825 static uintptr_t
1826 parseopt_pass2(Of1_desc *ofl, int argc, char **argv)
1827 {
1828     int    c, ndx = optind;
1830     while ((c = ld_getopt(ofl->ofl_lml, ndx, argc, argv)) != -1) {
1831         ifl_desc    *ifl;
1832         Sym_desc    *sdp;
1834         switch (c) {
1835             case 'l':
1836                 DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
1837                     optarg));
1838                 if (ld_find_library(optarg, ofl) == S_ERROR)
1839                     return (S_ERROR);
1840                 break;
1841             case 'B':
1842                 DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
1843                     optarg));
1844                 if (strcmp(optarg,
1845                     MSG_ORIG(MSG_STR_LD_DYNAMIC)) == 0) {
1846                     if (ofl->ofl_flags & FLG_OF_DYNAMIC)
1847                         ofl->ofl_flags |=
1848                             FLG_OF_DYNLIBS;
1849                 } else if (strcmp(optarg,
1850                     MSG_ORIG(MSG_ARG_STATIC)) == 0)

```

```

1856         ofl->ofl_flags &= ~FLG_OF_DYNLIBS;
1857         break;
1858     case 'L':
1859         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
1860             optarg));
1861         if (ld_add_libdir(ofl, optarg) == S_ERROR)
1862             return (S_ERROR);
1863         break;
1864     case 'N':
1865         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
1866             optarg));
1867         /*
1868          * Record DT_NEEDED string
1869          */
1870         if (!(ofl->ofl_flags & FLG_OF_DYNAMIC))
1871             ld_eprintf(ofl, ERR_FATAL,
1872                 MSG_INTL(MSG_ARG_ST_INCOMP),
1873                 MSG_ORIG(MSG_ARG_CN));
1874         if (((ifl = libld_calloc(1,
1875             sizeof (If1_desc))) == NULL) ||
1876             (aplist_append(&ofl->ofl_sos, ifl,
1877                 AL_CNT_OF_LIBS) == NULL))
1878             return (S_ERROR);
1880         ifl->ifl_name = MSG_INTL(MSG_STR_COMMAND);
1881         ifl->ifl_soname = optarg;
1882         ifl->ifl_flags = (FLG_IF_NEEDSTR |
1883             FLG_IF_FILEREF | FLG_IF_DEPREQD);
1885         break;
1886     case 'D':
1887         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
1888             optarg));
1889         (void) dbg_setup(ofl, optarg, 3);
1890         break;
1891     case 'u':
1892         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
1893             optarg));
1894         if (ld_sym_add_u(optarg, ofl,
1895             MSG_STR_COMMAND) == (Sym_desc *)S_ERROR)
1896             return (S_ERROR);
1897         break;
1898     case 'z':
1899         DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
1900             optarg));
1901         if ((strcmp(optarg, MSG_ORIG(MSG_ARG_LD32),
1902             MSG_ARG_LD32_SIZE) == 0) ||
1903             (strcmp(optarg, MSG_ORIG(MSG_ARG_LD64),
1904             MSG_ARG_LD64_SIZE) == 0)) {
1905             if (createargv(ofl, 0) == S_ERROR)
1906                 return (S_ERROR);
1907         } else if (strcmp(optarg,
1908             MSG_ORIG(MSG_ARG_ALLEXTRT)) == 0) {
1909             ofl->ofl_flags1 |= FLG_OF1_ALLEXRT;
1910             ofl->ofl_flags1 &= ~FLG_OF1_WEAKEXT;
1911         } else if (strcmp(optarg,
1912             MSG_ORIG(MSG_ARG_WEAKEXT)) == 0) {
1913             ofl->ofl_flags1 |= FLG_OF1_WEAKEXT;
1914             ofl->ofl_flags1 &= ~FLG_OF1_ALLEXRT;
1915         } else if (strcmp(optarg,
1916             MSG_ORIG(MSG_ARG_DFLEXTRT)) == 0) {
1917             ofl->ofl_flags1 &=
1918                 ~(FLG_OF1_ALLEXRT |
1919                 FLG_OF1_WEAKEXT);
1920         } else if (strcmp(optarg,
1921             MSG_ORIG(MSG_ARG_DIRECT)) == 0) {

```



```

1922         ofl->ofl_flags1 |= FLG_OF1_ZDIRECT;
1923         ofl->ofl_guideflags |= FLG_OFG_NO_DB;
1924     } else if (strcmp(optarg,
1925         MSG_ORIG(MSG_ARG_NODIRECT)) == 0) {
1926         ofl->ofl_flags1 &= ~FLG_OF1_ZDIRECT;
1927         ofl->ofl_guideflags |= FLG_OFG_NO_DB;
1928     } else if (strcmp(optarg,
1929         MSG_ORIG(MSG_ARG_IGNORE)) == 0) {
1930         ofl->ofl_flags1 |= FLG_OF1_IGNORE;
1931     } else if (strcmp(optarg,
1932         MSG_ORIG(MSG_ARG_RECORD)) == 0) {
1933         ofl->ofl_flags1 &= ~FLG_OF1_IGNORE;
1934     } else if (strcmp(optarg,
1935         MSG_ORIG(MSG_ARG_LAZYLOAD)) == 0) {
1936         ofl->ofl_flags1 |= FLG_OF1_LAZYLD;
1937         ofl->ofl_guideflags |= FLG_OFG_NO_LAZY;
1938     } else if (strcmp(optarg,
1939         MSG_ORIG(MSG_ARG_NOLAZYLOAD)) == 0) {
1940         ofl->ofl_flags1 &= ~FLG_OF1_LAZYLD;
1941         ofl->ofl_guideflags |= FLG_OFG_NO_LAZY;
1942     } else if (strcmp(optarg,
1943         MSG_ORIG(MSG_ARG_GROUPEM)) == 0) {
1944         ofl->ofl_flags1 |= FLG_OF1_GRPPEM;
1945     } else if (strcmp(optarg,
1946         MSG_ORIG(MSG_ARG_NOGROUPEM)) == 0) {
1947         ofl->ofl_flags1 &= ~FLG_OF1_GRPPEM;
1948     } else if (strcmp(optarg,
1949         MSG_ORIG(MSG_ARG_INITARRAY),
1950         MSG_ARG_INITARRAY_SIZE) == 0) {
1951         if ((sdp = ld_sym_add_u(optarg +
1952             MSG_ARG_INITARRAY_SIZE, ofl,
1953             MSG_STR_COMMAND)) ==
1954             (Sym_desc *)S_ERROR) ||
1955             (aplist_append(&ofl->ofl_initarray,
1956                 sdp, AL_CNT_OF1_ARRAYS) == NULL))
1957             return (S_ERROR);
1958     } else if (strcmp(optarg,
1959         MSG_ORIG(MSG_ARG_FINIARRAY),
1960         MSG_ARG_FINIARRAY_SIZE) == 0) {
1961         if ((sdp = ld_sym_add_u(optarg +
1962             MSG_ARG_FINIARRAY_SIZE, ofl,
1963             MSG_STR_COMMAND)) ==
1964             (Sym_desc *)S_ERROR) ||
1965             (aplist_append(&ofl->ofl_finiarray,
1966                 sdp, AL_CNT_OF1_ARRAYS) == NULL))
1967             return (S_ERROR);
1968     } else if (strcmp(optarg,
1969         MSG_ORIG(MSG_ARG_PREINITARRAY),
1970         MSG_ARG_PREINITARRAY_SIZE) == 0) {
1971         if ((sdp = ld_sym_add_u(optarg +
1972             MSG_ARG_PREINITARRAY_SIZE, ofl,
1973             MSG_STR_COMMAND)) ==
1974             (Sym_desc *)S_ERROR) ||
1975             (aplist_append(&ofl->ofl_preiarray,
1976                 sdp, AL_CNT_OF1_ARRAYS) == NULL))
1977             return (S_ERROR);
1978     } else if (strcmp(optarg,
1979         MSG_ORIG(MSG_ARG_RTLDINFO),
1980         MSG_ARG_RTLDINFO_SIZE) == 0) {
1981         if ((sdp = ld_sym_add_u(optarg +
1982             MSG_ARG_RTLDINFO_SIZE, ofl,
1983             MSG_STR_COMMAND)) ==
1984             (Sym_desc *)S_ERROR) ||
1985             (aplist_append(&ofl->ofl_rtldinfo,
1986                 sdp, AL_CNT_OF1_ARRAYS) == NULL))
1987             return (S_ERROR);

```

```

1988     } else if (strcmp(optarg,
1989         MSG_ORIG(MSG_ARG_DTRACE),
1990         MSG_ARG_DTRACE_SIZE) == 0) {
1991         if ((sdp = ld_sym_add_u(optarg +
1992             MSG_ARG_DTRACE_SIZE, ofl,
1993             MSG_STR_COMMAND)) ==
1994             (Sym_desc *)S_ERROR)
1995             return (S_ERROR);
1996         ofl->ofl_dtracesym = sdp;
1997     } else if (strcmp(optarg,
1998         MSG_ORIG(MSG_ARG_RESCAN_NOW)) == 0) {
1999         if (ld_rescan_archives(ofl, 0, ndx) ==
2000             S_ERROR)
2001             return (S_ERROR);
2002     } else if (strcmp(optarg,
2003         MSG_ORIG(MSG_ARG_RESCAN_START)) == 0) {
2004         ofl->ofl_ars_gsndx = ofl->ofl_arscnt;
2005         ofl->ofl_ars_gsndx = ndx;
2006     } else if (strcmp(optarg,
2007         MSG_ORIG(MSG_ARG_RESCAN_END)) == 0) {
2008         if (ld_rescan_archives(ofl, 1, ndx) ==
2009             S_ERROR)
2010             return (S_ERROR);
2011     } else if (strcmp(optarg,
2012         MSG_ORIG(MSG_ARG_DEFERRED)) == 0) {
2013         ofl->ofl_flags1 |= FLG_OF1_DEFERRED;
2014     } else if (strcmp(optarg,
2015         MSG_ORIG(MSG_ARG_NODEFERRED)) == 0) {
2016         ofl->ofl_flags1 &= ~FLG_OF1_DEFERRED;
2017     }
2018     default:
2019         break;
2020 }
2021
2022 /*
2023  * Update the argument index for the next getopt() iteration.
2024  */
2025     ndx = optind;
2026 }
2027     return (1);
2028 }
2029
2030 /*
2031  *
2032  * Pass 1 -- process_flags: collects all options and sets flags
2033  */
2034     static uintptr_t
2035     process_flags_com(Of1_desc *ofl, int argc, char **argv, int *usage)
2036     {
2037         for (; optind < argc; optind++) {
2038             /*
2039              * If we detect some more options return to getopt().
2040              * Checking argv[optind][1] against null prevents a forever
2041              * loop if an unadorned '-' argument is passed to us.
2042              */
2043             while ((optind < argc) && (argv[optind][0] == '-') {
2044                 if (argv[optind][1] != '\0') {
2045                     if (parseopt_pass1(ofl, argc, argv,
2046                         usage) == S_ERROR)
2047                         return (S_ERROR);
2048                 } else if (++optind < argc)
2049                     continue;
2050             }
2051             if (optind >= argc)
2052                 break;
2053             ofl->ofl_objscnt++;

```

```

2054     }
2055
2056     /* Did an unterminated archive group run off the end? */
2057     if (ofl->ofl_ars_gsndx > 0) {
2058         ld_printf(ofl, ERR_FATAL, MSG_INTL(MSG_ARG_AR_GRP_BAD),
2059                 MSG_INTL(MSG_MARG_AR_GRP_START),
2060                 MSG_INTL(MSG_MARG_AR_GRP_END));
2061         return (S_ERROR);
2062     }
2063
2064     return (1);
2065 }
2066
2067 uintptr_t
2068 ld_process_flags(Ofl_desc *ofl, int argc, char **argv)
2069 {
2070     int     usage = 0;      /* Collect all argument errors before exit */
2071
2072     if (argc < 2) {
2073         usage_msg(FALSE);
2074         return (S_ERROR);
2075     }
2076
2077     /*
2078      * Option handling
2079      */
2080     opterr = 0;
2081     optind = 1;
2082     if (process_flags_com(ofl, argc, argv, &usage) == S_ERROR)
2083         return (S_ERROR);
2084
2085     /*
2086      * Having parsed everything, did we have any usage errors.
2087      */
2088     if (usage) {
2089         eprintf(ofl->ofl_lml, ERR_FATAL, MSG_INTL(MSG_ARG_USEHELP));
2090         return (S_ERROR);
2091     }
2092
2093     return (check_flags(ofl, argc));
2094 }
2095
2096 /*
2097  * Pass 2 -- process_files: skips the flags collected in pass 1 and processes
2098  * files.
2099  */
2100 static uintptr_t
2101 process_files_com(Ofl_desc *ofl, int argc, char **argv)
2102 {
2103     for (; optind < argc; optind++) {
2104         int     fd;
2105         uintptr_t  open_ret;
2106         char    *path;
2107         Rej_desc  rej = { 0 };
2108
2109         /*
2110          * If we detect some more options return to getopt().
2111          * Checking argv[optind][1] against null prevents a forever
2112          * loop if an unadorned '-' argument is passed to us.
2113          */
2114         while ((optind < argc) && (argv[optind][0] == '-')) {
2115             if (argv[optind][1] != '\0') {
2116                 if (parseopt_pass2(ofl, argc, argv) == S_ERROR)
2117                     return (S_ERROR);
2118             } else if (++optind < argc)
2119                 continue;

```

```

2120     }
2121     if (optind >= argc)
2122         break;
2123
2124     path = argv[optind];
2125     if ((fd = open(path, O_RDONLY)) == -1) {
2126         int err = errno;
2127
2128         ld_printf(ofl, ERR_FATAL,
2129                 MSG_INTL(MSG_SYS_OPEN), path, strerror(err));
2130         continue;
2131     }
2132
2133     DBG_CALL(DBG_args_file(ofl->ofl_lml, optind, path));
2134
2135     open_ret = ld_process_open(path, path, &fd, ofl,
2136                               (FLG_IF_CMDLINE | FLG_IF_NEEDED), &rej, NULL);
2137     if (fd != -1)
2138         (void) close(fd);
2139     if (open_ret == S_ERROR)
2140         return (S_ERROR);
2141
2142     /*
2143      * Check for mismatched input.
2144      */
2145     if (rej.rej_type) {
2146         Conv_reject_desc_buf_t rej_buf;
2147
2148         ld_printf(ofl, ERR_FATAL,
2149                 MSG_INTL(reject[rej.rej_type]),
2150                 rej.rej_name ? rej.rej_name :
2151                 MSG_INTL(MSG_STR_UNKNOWN),
2152                 conv_reject_desc(&rej, &rej_buf,
2153                                 ld_targ.t_m.m_mach));
2154         return (1);
2155     }
2156 }
2157 return (1);
2158 }
2159
2160 uintptr_t
2161 ld_process_files(Ofl_desc *ofl, int argc, char **argv)
2162 {
2163     DBG_CALL(DBG_basic_files(ofl->ofl_lml));
2164
2165     /*
2166      * Process command line files (taking into account any applicable
2167      * preceding flags). Return if any fatal errors have occurred.
2168      */
2169     opterr = 0;
2170     optind = 1;
2171     if (process_files_com(ofl, argc, argv) == S_ERROR)
2172         return (S_ERROR);
2173     if (ofl->ofl_flags & FLG_OF_FATAL)
2174         return (1);
2175
2176     /*
2177      * Guidance: Use -B direct/nodirect or -z direct/nodirect.
2178      *
2179      * This is a backstop for the case where the link had no dependencies.
2180      * Otherwise, it will get caught by ld_process_ifl(). We need both,
2181      * because -z direct is positional, and its value at the time where
2182      * the first dependency is seen might be different than it is now.
2183      */
2184     if ((ofl->ofl_flags & FLG_OF_DYNAMIC) &&
2185         OFL_GUIDANCE(ofl, FLG_OFG_NO_DB)) {

```

```

2186         ld_eprintf(ofl, ERR_GUIDANCE, MSG_INTL(MSG_GUIDE_DIRECT));
2187         ofl->ofl_guideflags |= FLG_OFG_NO_DB;
2188     }
2190     /*
2191     * Now that all command line files have been processed see if there are
2192     * any additional 'needed' shared object dependencies.
2193     */
2194     if (ofl->ofl_soneed)
2195         if (ld_finish_libs(ofl) == S_ERROR)
2196             return (S_ERROR);
2198     /*
2199     * If rescanning archives is enabled, do so now to determine whether
2200     * there might still be members extracted to satisfy references from any
2201     * explicit objects. Continue until no new objects are extracted. Note
2202     * that this pass is carried out *after* processing any implicit objects
2203     * (above) as they may already have resolved any undefined references
2204     * from any explicit dependencies.
2205     */
2206     if (ofl->ofl_flags1 & FLG_OF1_RESCAN) {
2207         if (ld_rescan_archives(ofl, 0, argc) == S_ERROR)
2208             return (S_ERROR);
2209         if (ofl->ofl_flags & FLG_OF_FATAL)
2210             return (1);
2211     }
2213     /*
2214     * If debugging, provide statistics on each archives extraction, or flag
2215     * any archive that has provided no members. Note that this could be a
2216     * nice place to free up much of the archive infrastructure, as we've
2217     * extracted any members we need. However, as we presently don't free
2218     * anything under ld(1) there's not much point in proceeding further.
2219     */
2220     DBG_CALL(DBG_statistics_ar(ofl));
2222     /*
2223     * If any version definitions have been established, either via input
2224     * from a mapfile or from the input relocatable objects, make sure any
2225     * version dependencies are satisfied, and version symbols created.
2226     */
2227     if (ofl->ofl_verdesc)
2228         if (ld_vers_check_defs(ofl) == S_ERROR)
2229             return (S_ERROR);
2231     /*
2232     * If input section ordering was specified within some segment
2233     * using a mapfile, verify that the expected sections were seen.
2234     */
2235     if (ofl->ofl_flags & FLG_OF_IS_ORDER)
2236         ld_ent_check(ofl);
2238     return (1);
2239 }
2241 uintptr_t
2242 ld_init_strings(Of1_desc *ofl)
2243 {
2244     uint_t stflags;
2246     if (ofl->ofl_flags1 & FLG_OF1_NCSTTAB)
2247         stflags = 0;
2248     else
2249         stflags = FLG_STNEW_COMPRESS;
2251     if (((ofl->ofl_shdrsttab = st_new(stflags)) == NULL) ||

```

```

2252         ((ofl->ofl_strtab = st_new(stflags)) == NULL) ||
2253         ((ofl->ofl_dynstrtab = st_new(stflags)) == NULL))
2254             return (S_ERROR);
2256     return (0);
2257 }

```

```

*****
107960 Fri Mar 1 17:10:01 2019
new/usr/src/cmd/sgs/libld/common/files.c
code review from Robert
*****
_____unchanged_portion_omitted_____

3023 /*
3024 * Process the current input file.  There are basically three types of files
3025 * that come through here:
3026 *
3027 * - files explicitly defined on the command line (ie. foo.o or bar.so),
3028 *   in this case only the 'name' field is valid.
3029 *
3030 * - libraries determined from the -l command line option (ie. -lbar),
3031 *   in this case the 'soname' field contains the basename of the located
3032 *   file.
3033 *
3034 * Any shared object specified via the above two conventions must be recorded
3035 * as a needed dependency.
3036 *
3037 * - libraries specified as dependencies of those libraries already obtained
3038 *   via the command line (ie. bar.so has a DT_NEEDED entry of fred.so.1),
3039 *   in this case the 'soname' field contains either a full pathname (if the
3040 *   needed entry contained a '/'), or the basename of the located file.
3041 *   These libraries are processed to verify symbol binding but are not
3042 *   recorded as dependencies of the output file being generated.
3043 *
3044 * entry:
3045 *   name - File name
3046 *   soname - SONAME for needed sharable library, as described above
3047 *   fd - Open file descriptor
3048 *   elf - Open ELF handle
3049 *   flags - FLG_IF_ flags applicable to file
3050 *   ofl - Output file descriptor
3051 *   rej - Rejection descriptor used to record rejection reason
3052 *   ifl_ret - NULL, or address of pointer to receive reference to
3053 *             resulting input descriptor for file. If ifl_ret is non-NULL,
3054 *             the file cannot be an archive or it will be rejected.
3055 *
3056 * exit:
3057 *   If a error occurs in examining the file, S_ERROR is returned.
3058 *   If the file can be examined, but is not suitable, *rej is updated,
3059 *   and 0 is returned. If the file is acceptable, 1 is returned, and if
3060 *   ifl_ret is non-NULL, *ifl_ret is set to contain the pointer to the
3061 *   resulting input descriptor.
3062 */
3063 uintptr_t
3064 ld_process_ifl(const char *name, const char *soname, int fd, Elf *elf,
3065               Word flags, Ofld_desc *ofl, Rej_desc *rej, Ifl_desc **ifl_ret)
3066 {
3067     Ifl_desc      *ifl;
3068     Ehdr          *ehdr;
3069     uintptr_t     error = 0;
3070     struct stat   status;
3071     Ar_desc       *adp;
3072     Rej_desc      _rej;

3074     /*
3075     * If this file was not extracted from an archive obtain its device
3076     * information.  This will be used to determine if the file has already
3077     * been processed (rather than simply comparing filenames, the device
3078     * information provides a quicker comparison and detects linked files).
3079     */
3080     if (fd && ((flags & FLG_IF_EXTRACT) == 0))
3081         (void) fstat(fd, &status);

```

```

3082     else {
3083         status.st_dev = 0;
3084         status.st_ino = 0;
3085     }

3087     switch (elf_kind(elf)) {
3088     case ELF_K_AR:
3089         /*
3090         * If the caller has supplied a non-NULL ifl_ret, then
3091         * we cannot process archives, for there will be no
3092         * input file descriptor for us to return. In this case,
3093         * reject the attempt.
3094         */
3095         if (ifl_ret != NULL) {
3096             _rej.rej_type = SGS_REJ_ARCHIVE;
3097             _rej.rej_name = name;
3098             DBG_CALL(DBG_file_rejected(ofl->ofl_lml, &_rej,
3099                                     ld_targ.t_m.m_mach));
3100             if (rej->rej_type == 0) {
3101                 *rej = _rej;
3102                 rej->rej_name = strdup(_rej.rej_name);
3103             }
3104             return (0);
3105         }

3107         /*
3108         * Determine if we've already come across this archive file.
3109         */
3110         if (!(flags & FLG_IF_EXTRACT)) {
3111             Aliste idx;

3113             for (APLIST_TRAVERSE(ofl->ofl_ars, idx, adp)) {
3114                 if ((adp->ad_stdev != status.st_dev) ||
3115                     (adp->ad_stino != status.st_ino))
3116                     continue;

3118                 /*
3119                 * We've seen this file before so reuse the
3120                 * original archive descriptor and discard the
3121                 * new elf descriptor. Note that a file
3122                 * descriptor is unnecessary, as the file is
3123                 * already available in memory.
3124                 */
3125                 DBG_CALL(DBG_file_reuse(ofl->ofl_lml, name,
3126                                         adp->ad_name));
3127                 (void) elf_end(elf);
3128                 if (!ld_process_archive(name, -1, adp, ofl))
3129                     return (S_ERROR);
3130                 return (1);
3131             }
3132         }

3134         /*
3135         * As we haven't processed this file before establish a new
3136         * archive descriptor.
3137         */
3138         adp = ld_ar_setup(name, elf, ofl);
3139         if ((adp == NULL) || (adp == (Ar_desc *)S_ERROR))
3140             return ((uintptr_t)adp);
3141         adp->ad_stdev = status.st_dev;
3142         adp->ad_stino = status.st_ino;

3144         ld_sup_file(ofl, name, ELF_K_AR, flags, elf);

3146         /*
3147         * Indicate that the ELF descriptor no longer requires a file

```

```

3148     * descriptor by reading the entire file. The file is already
3149     * read via the initial mmap(2) behind elf_begin(3elf), thus
3150     * this operation is effectively a no-op. However, a side-
3151     * effect is that the internal file descriptor, maintained in
3152     * the ELF descriptor, is set to -1. This setting will not
3153     * be compared with any file descriptor that is passed to
3154     * elf_begin(), should this archive, or one of the archive
3155     * members, be processed again from the command line or
3156     * because of a -z rescan.
3157     */
3158     if (elf_cntl(elf, ELF_C_FDREAD) == -1) {
3159         ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_CNTL),
3160                 name);
3161         return (0);
3162     }

3164     if (!ld_process_archive(name, -1, adp, ofl))
3165         return (S_ERROR);
3166     return (1);

3168     case ELF_K_ELF:
3169         /*
3170          * Obtain the elf header so that we can determine what type of
3171          * elf ELF_K_ELF file this is.
3172          */
3173         if ((ehdr = elf_getehdr(elf)) == NULL) {
3174             int _class = gelf_getclass(elf);

3176             /*
3177              * This can fail for a number of reasons. Typically
3178              * the object class is incorrect (ie. user is building
3179              * 64-bit but managed to point at 32-bit libraries).
3180              * Other ELF errors can include a truncated or corrupt
3181              * file. Try to get the best error message possible.
3182              */
3183             if (ld_targ.t_m.m_class != _class) {
3184                 _rej.rej_type = SGS_REJ_CLASS;
3185                 _rej.rej_info = (uint_t)_class;
3186             } else {
3187                 _rej.rej_type = SGS_REJ_STR;
3188                 _rej.rej_str = elf_errmsg(-1);
3189             }
3190             _rej.rej_name = name;
3191             DBG_CALL(DBG_file_rejected(ofl->ofl_lml, &_rej,
3192                                     ld_targ.t_m.m_mach));
3193             if (rej->rej_type == 0) {
3194                 *_rej = _rej;
3195                 rej->rej_name = strdup(_rej.rej_name);
3196             }
3197             return (0);
3198         }

3200     if (_gelf_getdynval(elf, DT_SUNW_KMOD) > 0) {
3201     if (_gelf_getdynval(elf, DT_SUNW_KMOD) == 1) {
3202         _rej.rej_name = name;
3203         DBG_CALL(DBG_file_rejected(ofl->ofl_lml, &_rej,
3204                                 ld_targ.t_m.m_mach));
3205         _rej.rej_type = SGS_REJ_KMOD;
3206         _rej.rej_str = elf_errmsg(-1);
3207         _rej.rej_name = name;
3208     #endif /* ! codereview */

3209     if (rej->rej_type == 0) {
3210         *_rej = _rej;
3211         rej->rej_name = strdup(_rej.rej_name);
3212     }

```

```

3213         return (0);
3214     }

3216     /*
3217     * Determine if we've already come across this file.
3218     */
3219     if (!(flags & FLG_IF_EXTRACT)) {
3220         Aplist *apl;
3221         Aliste idx;

3223         if (ehdr->e_type == ET_REL)
3224             apl = ofl->ofl_objs;
3225         else
3226             apl = ofl->ofl_sos;

3228         /*
3229         * Traverse the appropriate file list and determine if
3230         * a dev/inode match is found.
3231         */
3232         for (APLIST_TRAVERSE(apl, idx, ifl)) {
3233             /*
3234              * Ifl_desc generated via -Nneed, therefore no
3235              * actual file behind it.
3236              */
3237             if (ifl->ifl_flags & FLG_IF_NEEDSTR)
3238                 continue;

3240             if ((ifl->ifl_stino != status.st_ino) ||
3241                 (ifl->ifl_stdev != status.st_dev))
3242                 continue;

3244             /*
3245              * Disregard (skip) this image.
3246              */
3247             DBG_CALL(DBG_file_skip(ofl->ofl_lml,
3248                                  ifl->ifl_name, name));
3249             (void) elf_end(elf);

3251             /*
3252              * If the file was explicitly defined on the
3253              * command line (this is always the case for
3254              * relocatable objects, and is true for shared
3255              * objects when they weren't specified via -l or
3256              * were dragged in as an implicit dependency),
3257              * then warn the user.
3258              */
3259             if ((flags & FLG_IF_CMDLINE) ||
3260                 (ifl->ifl_flags & FLG_IF_CMDLINE)) {
3261                 const char *errmsg;

3263                 /*
3264                  * Determine whether this is the same
3265                  * file name as originally encountered
3266                  * so as to provide the most
3267                  * descriptive diagnostic.
3268                  */
3269                 errmsg =
3270                     (strcmp(name, ifl->ifl_name) == 0) ?
3271                     MSG_INTL(MSG_FIL_MULINC_1) :
3272                     MSG_INTL(MSG_FIL_MULINC_2);
3273                 ld_eprintf(ofl, ERR_WARNING,
3274                         errmsg, name, ifl->ifl_name);
3275             }
3276             if (ifl_ret)
3277                 *ifl_ret = ifl;
3278             return (1);

```

```

3279     }
3280     }
3281
3282     /*
3283     * At this point, we know we need the file.  Establish an input
3284     * file descriptor and continue processing.
3285     */
3286     ifl = ifl_setup(name, ehdr, elf, flags, ofl, rej);
3287     if ((ifl == NULL) || (ifl == (ifl_desc *)S_ERROR))
3288         return ((uintptr_t)ifl);
3289     ifl->ifl_stdev = status.st_dev;
3290     ifl->ifl_stino = status.st_ino;
3291
3292     /*
3293     * If -ignore is in effect, mark this file as a potential
3294     * candidate (the files use isn't actually determined until
3295     * symbol resolution and relocation processing are completed).
3296     */
3297     if (ofl->ofl_flags1 & FLG_OF1_IGNORE)
3298         ifl->ifl_flags |= FLG_IF_IGNORE;
3299
3300     switch (ehdr->e_type) {
3301     case ET_REL:
3302         (*ld_targ.t_mr.mr_mach_eflags)(ehdr, ofl);
3303         error = process_elf(ifl, elf, ofl);
3304         break;
3305     case ET_DYN:
3306         if ((ofl->ofl_flags & FLG_OF_STATIC) ||
3307             !(ofl->ofl_flags & FLG_OF_DYNLIBS)) {
3308             ld_eprintf(ofl, ERR_FATAL,
3309                 MSG_INTL(MSG_FIL_SOINSTAT), name);
3310             return (0);
3311         }
3312
3313         /*
3314         * Record any additional shared object information.
3315         * If no soname is specified (eg. this file was
3316         * derived from a explicit filename declaration on the
3317         * command line, ie. bar.so) use the pathname.
3318         * This entry may be overridden if the files dynamic
3319         * section specifies an DT_SONAME value.
3320         */
3321         if (soname == NULL)
3322             ifl->ifl_soname = ifl->ifl_name;
3323         else
3324             ifl->ifl_soname = soname;
3325
3326         /*
3327         * If direct bindings, lazy loading, group permissions,
3328         * or deferred dependencies need to be established, mark
3329         * this object.
3330         */
3331         if (ofl->ofl_flags1 & FLG_OF1_ZDIRECT)
3332             ifl->ifl_flags |= FLG_IF_DIRECT;
3333         if (ofl->ofl_flags1 & FLG_OF1_LAZYLD)
3334             ifl->ifl_flags |= FLG_IF_LAZYLD;
3335         if (ofl->ofl_flags1 & FLG_OF1_GRPPRM)
3336             ifl->ifl_flags |= FLG_IF_GRPPRM;
3337         if (ofl->ofl_flags1 & FLG_OF1_DEFERRED)
3338             ifl->ifl_flags |=
3339                 (FLG_IF_LAZYLD | FLG_IF_DEFERRED);
3340
3341         error = process_elf(ifl, elf, ofl);
3342
3343         /*
3344         * Determine whether this dependency requires a syminfo.

```

```

3345     */
3346     if (ifl->ifl_flags & MSK_IF_SYMINFO)
3347         ofl->ofl_flags |= FLG_OF_SYMINFO;
3348
3349     /*
3350     * Guidance: Use -z lazyload/nolazyload.
3351     * libc is exempt from this advice, because it cannot
3352     * be lazy loaded, and requests to do so are ignored.
3353     */
3354     if (OFL_GUIDANCE(ofl, FLG_OFG_NO_LAZY) &&
3355         ((ifl->ifl_flags & FLG_IF_RTLDINF) == 0)) {
3356         ld_eprintf(ofl, ERR_GUIDANCE,
3357             MSG_INTL(MSG_GUIDE_LAZYLOAD));
3358         ofl->ofl_guideflags |= FLG_OFG_NO_LAZY;
3359     }
3360
3361     /*
3362     * Guidance: Use -B direct/nodirect or
3363     * -z direct/nodirect.
3364     */
3365     if (OFL_GUIDANCE(ofl, FLG_OFG_NO_DB)) {
3366         ld_eprintf(ofl, ERR_GUIDANCE,
3367             MSG_INTL(MSG_GUIDE_DIRECT));
3368         ofl->ofl_guideflags |= FLG_OFG_NO_DB;
3369     }
3370
3371     break;
3372 default:
3373     (void) elf_errno();
3374     _rej.rej_type = SGS_REJ_UNKFILE;
3375     _rej.rej_name = name;
3376     DBG_CALL(DBG_file_rejected(ofl->ofl_lml, &_rej,
3377         ld_targ.t.m.m_mach));
3378     if (rej->rej_type == 0) {
3379         *_rej = _rej;
3380         rej->rej_name = strdup(_rej.rej_name);
3381     }
3382     return (0);
3383 }
3384 break;
3385 default:
3386     (void) elf_errno();
3387     _rej.rej_type = SGS_REJ_UNKFILE;
3388     _rej.rej_name = name;
3389     DBG_CALL(DBG_file_rejected(ofl->ofl_lml, &_rej,
3390         ld_targ.t.m.m_mach));
3391     if (rej->rej_type == 0) {
3392         *_rej = _rej;
3393         rej->rej_name = strdup(_rej.rej_name);
3394     }
3395     return (0);
3396 }
3397 if ((error == 0) || (error == S_ERROR))
3398     return (error);
3399
3400 if (ifl_ret)
3401     *ifl_ret = ifl;
3402 return (1);
3403 }
3404
3405 /*
3406 * Having successfully opened a file, set up the necessary elf structures to
3407 * process it further.  This small section of processing is slightly different
3408 * from the elf initialization required to process a relocatable object from an
3409 * archive (see libs.c: ld_process_archive()).
3410 */

```

```

3411 uintptr_t
3412 ld_process_open(const char *opath, const char *ofile, int *fd, Ofld_desc *ofl,
3413               Word flags, Rej_desc *rej, Ifld_desc **ifld_ret)
3414 {
3415     Elf          *elf;
3416     const char   *npath = opath;
3417     const char   *nfile = ofile;
3418
3419     if ((elf = elf_begin(*fd, ELF_C_READ, NULL)) == NULL) {
3420         ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_BEGIN), npath);
3421         return (0);
3422     }
3423
3424     /*
3425      * Determine whether the support library wishes to process this open.
3426      * The support library may return:
3427      * . a different ELF descriptor (in which case they should have
3428      *   closed the original)
3429      * . a different file descriptor (in which case they should have
3430      *   closed the original)
3431      * . a different path and file name (presumably associated with
3432      *   a different file descriptor)
3433      *
3434      * A file descriptor of -1, or and ELF descriptor of zero indicates
3435      * the file should be ignored.
3436      */
3437     ld_sup_open(ofl, &npath, &nfile, fd, flags, &elf, NULL, 0,
3438               elf_kind(elf));
3439
3440     if ((*fd == -1) || (elf == NULL))
3441         return (0);
3442
3443     return (ld_process_ifld(npath, nfile, *fd, elf, flags, ofl, rej,
3444                          ifld_ret));
3445 }
3446
3447 /*
3448  * Having successfully mapped a file, set up the necessary elf structures to
3449  * process it further. This routine is patterned after ld_process_open() and
3450  * is only called by ld.so.1(1) to process a relocatable object.
3451  */
3452 Ifld_desc *
3453 ld_process_mem(const char *path, const char *file, char *addr, size_t size,
3454               Ofld_desc *ofl, Rej_desc *rej)
3455 {
3456     Elf          *elf;
3457     uintptr_t    open_ret;
3458     Ifld_desc    *ifld;
3459
3460     if ((elf = elf_memory(addr, size)) == NULL) {
3461         ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_MEMORY), path);
3462         return (0);
3463     }
3464
3465     open_ret = ld_process_ifld(path, file, 0, elf, 0, ofl, rej, &ifld);
3466     if (open_ret != 1)
3467         return ((Ifld_desc *) open_ret);
3468     return (ifld);
3469 }
3470
3471 /*
3472  * Process a required library (i.e. the dependency of a shared object).
3473  * Combine the directory and filename, check the resultant path size, and try
3474  * opening the pathname.
3475  */
3476 static Ifld_desc *

```

```

3477 process_req_lib(Sdf_desc *sdf, const char *dir, const char *file,
3478               Ofld_desc *ofl, Rej_desc *rej)
3479 {
3480     size_t      dlen, plen;
3481     int         fd;
3482     char        path[PATH_MAX];
3483     const char  *_dir = dir;
3484
3485     /*
3486      * Determine the sizes of the directory and filename to insure we don't
3487      * exceed our buffer.
3488      */
3489     if ((dlen = strlen(dir)) == 0) {
3490         _dir = MSG_ORIG(MSG_STR_DOT);
3491         dlen = 1;
3492     }
3493     dlen++;
3494     plen = dlen + strlen(file) + 1;
3495     if (plen > PATH_MAX) {
3496         ld_eprintf(ofl, ERR_FATAL, MSG_INTL(MSG_FIL_PTHTOLONG),
3497                 _dir, file);
3498         return (0);
3499     }
3500
3501     /*
3502      * Build the entire pathname and try and open the file.
3503      */
3504     (void) strcpy(path, _dir);
3505     (void) strcat(path, MSG_ORIG(MSG_STR_SLASH));
3506     (void) strcat(path, file);
3507     DBG_CALL(DBG_libs_req(ofl->ofl_lml, sdf->sdf_name,
3508                        sdf->sdf_rfile, path));
3509
3510     if ((fd = open(path, O_RDONLY)) == -1)
3511         return (0);
3512     else {
3513         uintptr_t    open_ret;
3514         Ifld_desc    *ifld;
3515         char         *_path;
3516
3517         if ((_path = libld_malloc(strlen(path) + 1)) == NULL)
3518             return ((Ifld_desc *)S_ERROR);
3519         (void) strcpy(_path, path);
3520         open_ret = ld_process_open(_path, &_path[dlen], &fd, ofl,
3521                                   0, rej, &ifld);
3522         if (fd != -1)
3523             (void) close(fd);
3524         if (open_ret != 1)
3525             return ((Ifld_desc *)open_ret);
3526         return (ifld);
3527     }
3528 }
3529
3530 /*
3531  * Finish any library processing. Walk the list of so's that have been listed
3532  * as "included" by shared objects we have previously processed. Examine them,
3533  * without adding them as explicit dependents of this program, in order to
3534  * complete our symbol definition process. The search path rules are:
3535  *
3536  * - use any user supplied paths, i.e. LD_LIBRARY_PATH and -L, then
3537  *
3538  * - use any RPATH defined within the parent shared object, then
3539  *
3540  * - use the default directories, i.e. LIBPATH or -YP.
3541  */
3542 uintptr_t

```

```

3543 ld_finish_libs(Of1_desc *of1)
3544 {
3545     Aliste      idx1;
3546     Sdf_desc    *sdf;
3547     Rej_desc    rej = { 0 };
3548
3549     /*
3550      * Make sure we are back in dynamic mode.
3551      */
3552     of1->of1_flags |= FLG_OF_DYNLIBS;
3553
3554     for (APLIST_TRAVERSE(of1->of1_soneed, idx1, sdf)) {
3555         Aliste      idx2;
3556         char        *path, *slash = NULL;
3557         int         fd;
3558         Ifl_desc    *ifl;
3559         char        *file = (char *)sdf->sdf_name;
3560
3561         /*
3562          * See if this file has already been processed. At the time
3563          * this implicit dependency was determined there may still have
3564          * been more explicit dependencies to process. Note, if we ever
3565          * do parse the command line three times we would be able to
3566          * do all this checking when processing the dynamic section.
3567          */
3568         if (sdf->sdf_file)
3569             continue;
3570
3571         for (APLIST_TRAVERSE(of1->of1_sos, idx2, ifl)) {
3572             if (!(ifl->ifl_flags & FLG_IF_NEEDSTR) &&
3573                 (strcmp(file, ifl->ifl_soname) == 0)) {
3574                 sdf->sdf_file = ifl;
3575                 break;
3576             }
3577         }
3578         if (sdf->sdf_file)
3579             continue;
3580
3581         /*
3582          * If the current path name element embeds a "/", then it's to
3583          * be taken "as is", with no searching involved. Process all
3584          * "/" occurrences, so that we can deduce the base file name.
3585          */
3586         for (path = file; *path; path++) {
3587             if (*path == '/')
3588                 slash = path;
3589         }
3590         if (slash) {
3591             DBG_CALL(DBG_libs_req(of1->of1_lml, sdf->sdf_name,
3592                                 sdf->sdf_rfile, file));
3593             if ((fd = open(file, O_RDONLY)) == -1) {
3594                 ld_eprintf(of1, ERR_WARNING,
3595                             MSG_INTL(MSG_FIL_NOTFOUND), file,
3596                             sdf->sdf_rfile);
3597             } else {
3598                 uintptr_t    open_ret;
3599                 Rej_desc      _rej = { 0 };
3600
3601                 open_ret = ld_process_open(file, ++slash,
3602                                           &fd, of1, 0, &rej, &ifl);
3603                 if (fd != -1)
3604                     (void) close(fd);
3605                 if (open_ret == S_ERROR)
3606                     return (S_ERROR);
3607
3608                 if (_rej.rej_type) {

```

```

3609         Conv_reject_desc_buf_t rej_buf;
3610
3611         ld_eprintf(of1, ERR_WARNING,
3612                   MSG_INTL(reject[_rej.rej_type]),
3613                   _rej.rej_name ? rej.rej_name :
3614                   MSG_INTL(MSG_STR_UNKNOWN),
3615                   conv_reject_desc(&rej, &rej_buf,
3616                                   ld_targ.t_m.m_mach));
3617     } else
3618         sdf->sdf_file = ifl;
3619     }
3620     continue;
3621 }
3622
3623 /*
3624  * Now search for this file in any user defined directories.
3625  */
3626 for (APLIST_TRAVERSE(of1->of1_ulibdirs, idx2, path)) {
3627     Rej_desc      _rej = { 0 };
3628
3629     ifl = process_req_lib(sdf, path, file, of1, &rej);
3630     if (ifl == (Ifl_desc *)S_ERROR) {
3631         return (S_ERROR);
3632     }
3633     if (_rej.rej_type) {
3634         if (rej.rej_type == 0) {
3635             rej = _rej;
3636             rej.rej_name = strdup(_rej.rej_name);
3637         }
3638     }
3639     if (ifl) {
3640         sdf->sdf_file = ifl;
3641         break;
3642     }
3643 }
3644 if (sdf->sdf_file)
3645     continue;
3646
3647 /*
3648  * Next use the local rules defined within the parent shared
3649  * object.
3650  */
3651 if (sdf->sdf_rpath != NULL) {
3652     char          *rpath, *next;
3653
3654     rpath = libld_malloc(strlen(sdf->sdf_rpath) + 1);
3655     if (rpath == NULL)
3656         return (S_ERROR);
3657     (void) strcpy(rpath, sdf->sdf_rpath);
3658     DBG_CALL(DBG_libs_path(of1->of1_lml, rpath,
3659                           LA_SER_RUNPATH, sdf->sdf_rfile));
3660     if ((path = strtok_r(rpath,
3661                         MSG_ORIG(MSG_STR_COLON), &next)) != NULL) {
3662         do {
3663             Rej_desc      _rej = { 0 };
3664
3665             path = expand(sdf->sdf_rfile, path,
3666                          &next);
3667
3668             ifl = process_req_lib(sdf, path,
3669                                 file, of1, &rej);
3670             if (ifl == (Ifl_desc *)S_ERROR) {
3671                 return (S_ERROR);
3672             }
3673             if ((_rej.rej_type) &&
3674                 (rej.rej_type == 0)) {

```



```

3675         rej = _rej;
3676         rej.rej_name =
3677             strdup(_rej.rej_name);
3678     }
3679     if (ifl) {
3680         sdf->sdf_file = ifl;
3681         break;
3682     }
3683     } while ((path = strtok_r(NULL,
3684         MSG_ORIG(MSG_STR_COLON), &next) != NULL);
3685     }
3686 }
3687 if (sdf->sdf_file)
3688     continue;
3689
3690 /*
3691  * Finally try the default library search directories.
3692  */
3693 for (APLIST_TRAVERSE(ofl->ofl_dlibdirs, idx2, path)) {
3694     Rej_desc     _rej = { 0 };
3695
3696     ifl = process_req_lib(sdf, path, file, ofl, &rej);
3697     if (ifl == (Ifldesc *)S_ERROR) {
3698         return (S_ERROR);
3699     }
3700     if (_rej.rej_type) {
3701         if (rej.rej_type == 0) {
3702             rej = _rej;
3703             rej.rej_name = strdup(_rej.rej_name);
3704         }
3705     }
3706     if (ifl) {
3707         sdf->sdf_file = ifl;
3708         break;
3709     }
3710 }
3711 if (sdf->sdf_file)
3712     continue;
3713
3714 /*
3715  * If we've got this far we haven't found the shared object.
3716  * If an object was found, but was rejected for some reason,
3717  * print a diagnostic to that effect, otherwise generate a
3718  * generic "not found" diagnostic.
3719  */
3720 if (rej.rej_type) {
3721     Conv_reject_desc_buf_t rej_buf;
3722
3723     ld_eprintf(ofl, ERR_WARNING,
3724         MSG_INTL(reject[rej.rej_type]),
3725         rej.rej_name ? rej.rej_name :
3726         MSG_INTL(MSG_STR_UNKNOWN),
3727         conv_reject_desc(&rej, &rej_buf,
3728             ld_targ.t.m.m_mach));
3729 } else {
3730     ld_eprintf(ofl, ERR_WARNING,
3731         MSG_INTL(MSG_FIL_NOTFOUND), file, sdf->sdf_rfile);
3732 }
3733 }
3734
3735 /*
3736  * Finally, now that all objects have been input, make sure any version
3737  * requirements have been met.
3738  */
3739 return (ld_vers_verify(ofl));
3740 }

```

```

*****
96513 Fri Mar 1 17:10:02 2019
new/usr/src/cmd/sgs/libld/common/sections.c
code review from Robert
*****
_____unchanged_portion_omitted_____

927 /*
928 * Make the dynamic section. Calculate the size of any strings referenced
929 * within this structure, they will be added to the global string table
930 * (.dynstr). This routine should be called before make_dynstr().
931 *
932 * This routine must be maintained in parallel with update_odynamic()
933 * in update.c
934 */
935 static uintptr_t
936 make_dynamic(Of1_desc *of1)
937 {
938     Shdr      *shdr;
939     Os_desc   *osp;
940     Elf_Data  *data;
941     Is_desc   *isec;
942     size_t    cnt = 0;
943     Aliste    idx;
944     Ifl_desc  *ifl;
945     Sym_desc  *sdp;
946     size_t    size;
947     Str_tbl   *strtbl;
948     ofl_flag_t flags = of1->ofl_flags;
949     int       not_relobj = !(flags & FLG_OF_RELOBJ);
950     int       unused = 0;

952 /*
953  * Select the required string table.
954  */
955 if (OFL_IS_STATIC_OBJ(of1))
956     strtbl = of1->ofl_strtab;
957 else
958     strtbl = of1->ofl_dynstrtab;

960 /*
961  * Only a limited subset of DT_entries apply to relocatable
962  * objects. See the comment at the head of update_odynamic() in
963  * update.c for details.
964  */
965 if (new_section(of1, SHT_DYNAMIC, MSG_ORIG(MSG_SCN_DYNAMIC), 0,
966               &isec, &shdr, &data) == S_ERROR)
967     return (S_ERROR);

969 /*
970  * new_section() does not set SHF_ALLOC. If we're building anything
971  * besides a relocatable object, then the .dynamic section should
972  * reside in allocatable memory.
973  */
974 if (not_relobj)
975     shdr->sh_flags |= SHF_ALLOC;

977 /*
978  * new_section() does not set SHF_WRITE. If we're building an object
979  * that specifies an interpreter, then a DT_DEBUG entry is created,
980  * which is initialized to the applications link-map list at runtime.
981  */
982 if (of1->ofl_osinterp)
983     shdr->sh_flags |= SHF_WRITE;

985     osp = of1->ofl_osdynamic =

```

```

986     ld_place_section(of1, isec, NULL, ld_targ.t_id.id_dynamic, NULL);

988 /*
989  * Reserve entries for any needed dependencies.
990  */
991 for (APLIST_TRAVERSE(of1->ofl_sos, idx, ifl)) {
992     if (!(ifl->ifl_flags & (FLG_IF_NEEDED | FLG_IF_NEEDSTR)))
993         continue;

995 /*
996  * If this dependency didn't satisfy any symbol references,
997  * generate a debugging diagnostic (ld(1) -Dunused can be used
998  * to display these). If this is a standard needed dependency,
999  * and -z ignore is in effect, drop the dependency. Explicitly
1000  * defined dependencies (i.e., -N dep) don't get dropped, and
1001  * are flagged as being required to simplify update_odynamic()
1002  * processing.
1003  */
1004 if ((ifl->ifl_flags & FLG_IF_NEEDSTR) ||
1005     ((ifl->ifl_flags & FLG_IF_DEPREQD) == 0)) {
1006     if (unused++ == 0)
1007         DBG_CALL(DBG_util_nl(of1->ofl_lml, DBG_NL_STD));
1008     DBG_CALL(DBG_unused_file(of1->ofl_lml, ifl->ifl_soname,
1009                            (ifl->ifl_flags & FLG_IF_NEEDSTR), 0));

1011 /*
1012  * Guidance: Remove unused dependency.
1013  *
1014  * If -z ignore is in effect, this warning is not
1015  * needed because we will quietly remove the unused
1016  * dependency.
1017  */
1018 if (OFL_GUIDANCE(of1, FLG_OFG_NO_UNUSED) &&
1019     ((ifl->ifl_flags & FLG_IF_IGNORE) == 0))
1020     ld_eprintf(of1, ERR_GUIDANCE,
1021              MSG_INTL(MSG_GUIDE_UNUSED),
1022              ifl->ifl_soname);

1024     if (ifl->ifl_flags & FLG_IF_NEEDSTR)
1025         ifl->ifl_flags |= FLG_IF_DEPREQD;
1026     else if (ifl->ifl_flags & FLG_IF_IGNORE)
1027         continue;
1028 }

1030 /*
1031  * If this object requires a DT_POSFLAG_1 entry, reserve it.
1032  */
1033 if ((ifl->ifl_flags & MSK_IF_POSFLAG1) && not_relobj)
1034     cnt++;

1036 if (st_insert(strtbl, ifl->ifl_soname) == -1)
1037     return (S_ERROR);
1038 cnt++;

1040 /*
1041  * If the associated entry contains the $ORIGIN token make sure
1042  * the associated DT_1_FLAGS entry is created.
1043  */
1044 if (strstr(ifl->ifl_soname, MSG_ORIG(MSG_STR_ORIGIN))) {
1045     ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
1046     ofl->ofl_dtflags |= DF_ORIGIN;
1047 }
1048 }

1050 if (unused)
1051     DBG_CALL(DBG_util_nl(of1->ofl_lml, DBG_NL_STD));

```

```

1053     if (not_relobj) {
1054         /*
1055          * Reserve entries for any per-symbol auxiliary/filter strings.
1056          */
1057         cnt += alist_nitems(ofl->ofl_dtsfltrs);

1059         /*
1060          * Reserve entries for _init() and _fini() section addresses.
1061          */
1062         if (((sdp = ld_sym_find(MSG_ORIG(MSG_SYM_INIT_U),
1063             SYM_NOHASH, NULL, ofl)) != NULL) &&
1064             (sdp->sd_ref == REF_REL_NEED) &&
1065             (sdp->sd_sym->st_shndx != SHN_UNDEF)) {
1066             sdp->sd_flags |= FLG_SY_UPREQD;
1067             cnt++;
1068         }
1069         if (((sdp = ld_sym_find(MSG_ORIG(MSG_SYM FINI_U),
1070             SYM_NOHASH, NULL, ofl)) != NULL) &&
1071             (sdp->sd_ref == REF_REL_NEED) &&
1072             (sdp->sd_sym->st_shndx != SHN_UNDEF)) {
1073             sdp->sd_flags |= FLG_SY_UPREQD;
1074             cnt++;
1075         }
1076     }

1077     /*
1078     * Reserve entries for any soname, filter name (shared libs
1079     * only), run-path pointers, cache names and audit requirements.
1080     */
1081     if (ofl->ofl_soname) {
1082         cnt++;
1083         if (st_insert(strtbl, ofl->ofl_soname) == -1)
1084             return (S_ERROR);
1085     }
1086     if (ofl->ofl_filtees) {
1087         cnt++;
1088         if (st_insert(strtbl, ofl->ofl_filtees) == -1)
1089             return (S_ERROR);
1090     }

1091     /*
1092     * If the filtees entry contains the $ORIGIN token
1093     * make sure the associated DT_1_FLAGS entry is created.
1094     */
1095     if (strstr(ofl->ofl_filtees,
1096         MSG_ORIG(MSG_STR_ORIGIN))) {
1097         ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
1098         ofl->ofl_dtflags |= DF_ORIGIN;
1099     }
1100 }

1103     if (ofl->ofl_rpath) {
1104         cnt += 2; /* DT_RPATH & DT_RUNPATH */
1105         if (st_insert(strtbl, ofl->ofl_rpath) == -1)
1106             return (S_ERROR);

1108         /*
1109         * If the rpath entry contains the $ORIGIN token make sure
1110         * the associated DT_1_FLAGS entry is created.
1111         */
1112         if (strstr(ofl->ofl_rpath, MSG_ORIG(MSG_STR_ORIGIN))) {
1113             ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
1114             ofl->ofl_dtflags |= DF_ORIGIN;
1115         }
1116     }

```

```

1118     if (not_relobj) {
1119         Aliste idx;
1120         Sg_desc *sgp;

1122         if (ofl->ofl_config) {
1123             cnt++;
1124             if (st_insert(strtbl, ofl->ofl_config) == -1)
1125                 return (S_ERROR);

1127             /*
1128             * If the config entry contains the $ORIGIN token
1129             * make sure the associated DT_1_FLAGS entry is created.
1130             */
1131             if (strstr(ofl->ofl_config, MSG_ORIG(MSG_STR_ORIGIN))) {
1132                 ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
1133                 ofl->ofl_dtflags |= DF_ORIGIN;
1134             }
1135         }
1136         if (ofl->ofl_depaudit) {
1137             cnt++;
1138             if (st_insert(strtbl, ofl->ofl_depaudit) == -1)
1139                 return (S_ERROR);
1140         }
1141         if (ofl->ofl_audit) {
1142             cnt++;
1143             if (st_insert(strtbl, ofl->ofl_audit) == -1)
1144                 return (S_ERROR);
1145         }
1146     }

1147     /*
1148     * Reserve entries for the DT_HASH, DT_STRTAB, DT_STRSZ,
1149     * DT_SYMTAB, DT_SYMENT, and DT_CHECKSUM.
1150     */
1151     cnt += 6;

1153     /*
1154     * If we are including local functions at the head of
1155     * the dynsym, then also reserve entries for DT_SUNW_SYMTAB
1156     * and DT_SUNW_SYMSZ.
1157     */
1158     if (OFL_ALLOW_LDYSYM(ofl))
1159         cnt += 2;

1161     if ((ofl->ofl_dynsymsortcnt > 0) ||
1162         (ofl->ofl_dyntlssortcnt > 0))
1163         cnt++; /* DT_SUNW_SORTENT */

1165     if (ofl->ofl_dynsymsortcnt > 0)
1166         cnt += 2; /* DT_SUNW_[SYMSORT|SYMSORTSZ] */

1168     if (ofl->ofl_dyntlssortcnt > 0)
1169         cnt += 2; /* DT_SUNW_[TLSSORT|TLSSORTSZ] */

1171     if ((flags & (FLG_OF_VERDEF | FLG_OF_NOVERSEC)) ==
1172         FLG_OF_VERDEF)
1173         cnt += 2; /* DT_VERDEF & DT_VERDEFNUM */

1175     if ((flags & (FLG_OF_VERNEED | FLG_OF_NOVERSEC)) ==
1176         FLG_OF_VERNEED)
1177         cnt += 2; /* DT_VERNEED & DT_VERNEEDNUM */

1179     if ((flags & FLG_OF_COMREL) && ofl->ofl_relocrelcnt)
1180         cnt++; /* DT_RELACOUNT */

1182     if (flags & FLG_OF_TEXTREL) /* DT_TEXTREL */
1183         cnt++;

```

```

1185         if (ofl->ofl_osfiniarray)      /* DT_FINI_ARRAY */
1186             cnt += 2;                  /* DT_FINI_ARRAYSZ */

1188         if (ofl->ofl_osinitarray)      /* DT_INIT_ARRAY */
1189             cnt += 2;                  /* DT_INIT_ARRAYSZ */

1191         if (ofl->ofl_ospreinitarray)   /* DT_PREINIT_ARRAY & */
1192             cnt += 2;                  /* DT_PREINIT_ARRAYSZ */

1194         /*
1195          * If we have plt's reserve a DT_PLTRELSZ, DT_PLTREL and
1196          * DT_JMPREL.
1197          */
1198         if (ofl->ofl_pltcnt)
1199             cnt += 3;

1201         /*
1202          * If plt padding is needed (Sparcv9).
1203          */
1204         if (ofl->ofl_pltpad)
1205             cnt += 2;                  /* DT_PLTPAD & DT_PLTPADSZ */

1207         /*
1208          * If we have any relocations reserve a DT_REL, DT_RELSZ and
1209          * DT_RELENT entry.
1210          */
1211         if (ofl->ofl_relocsz)
1212             cnt += 3;

1214         /*
1215          * If a syminfo section is required create DT_SYMINFO,
1216          * DT_SYMINSZ, and DT_SYMINENT entries.
1217          */
1218         if (flags & FLG_OF_SYMINFO)
1219             cnt += 3;

1221         /*
1222          * If there are any partially initialized sections allocate
1223          * DT_MOVETAB, DT_MOVESZ and DT_MOVEENT.
1224          */
1225         if (ofl->ofl_osmove)
1226             cnt += 3;

1228         /*
1229          * Allocate one DT_REGISTER entry for every register symbol.
1230          */
1231         cnt += ofl->ofl_regsymcnt;

1233         /*
1234          * Reserve a entry for each '-zrtldinfo=...' specified
1235          * on the command line.
1236          */
1237         for (APLIST_TRAVERSE(ofl->ofl_rtlldinfo, idx, sdp))
1238             cnt++;

1240         /*
1241          * The following entry should only be placed in a segment that
1242          * is writable.
1243          */
1244         if (((sgp = osp->os_sgdesc) != NULL) &&
1245             (sgp->sg_phdr.p_flags & PF_W) && ofl->ofl_osinterp)
1246             cnt++;                  /* DT_DEBUG */

1248         /*
1249          * Capabilities require a .dynamic entry for the .SUNW_cap

```

```

1250         * section.
1251         */
1252         if (ofl->ofl_oscapp)
1253             cnt++;                  /* DT_SUNW_CAP */

1255         /*
1256          * Symbol capabilities require a .dynamic entry for the
1257          * .SUNW_capinfo section.
1258          */
1259         if (ofl->ofl_oscappinfo)
1260             cnt++;                  /* DT_SUNW_CAPINFO */

1262         /*
1263          * Capabilities chain information requires a .SUNW_capchain
1264          * entry (DT_SUNW_CAPCHAIN), entry size (DT_SUNW_CAPCHAINENT),
1265          * and total size (DT_SUNW_CAPCHAINSZ).
1266          */
1267         if (ofl->ofl_oscappchain)
1268             cnt += 3;

1270         if (flags & FLG_OF_SYMBOLIC)
1271             cnt++;                  /* DT_SYMBOLIC */

1273         if (ofl->ofl_aslr != 0)
1274             cnt++;                  /* DT_SUNW_ASLR */
1275     }

1277     /* DT_SUNW_KMOD */
1278 #endif /* ! codereview */
1279     if (ofl->ofl_flags & FLG_OF_KMOD)
1280         cnt++;

1282     /*
1283      * Account for Architecture dependent .dynamic entries, and defaults.
1284      */
1285     (*ld_targ.t_mr.mr_mach_make_dynamic)(ofl, &cnt);

1287     /*
1288      * DT_FLAGS, DT_FLAGS_1, DT_SUNW_STRPAD, and DT_NULL. Also,
1289      * allow room for the unused extra DT_NULLs. These are included
1290      * to allow an ELF editor room to add items later.
1291      */
1292     cnt += 4 + DYNAMIC_EXTRAELTS;

1294     /*
1295      * DT_SUNW_LDMACH. Used to hold the ELF machine code of the
1296      * linker that produced the output object. This information
1297      * allows us to determine whether a given object was linked
1298      * natively, or by a linker running on a different type of
1299      * system. This information can be valuable if one suspects
1300      * that a problem might be due to alignment or byte order issues.
1301      */
1302     cnt++;

1304     /*
1305      * Determine the size of the section from the number of entries.
1306      */
1307     size = cnt * (size_t)shdr->sh_entsize;

1309     shdr->sh_size = (Xword)size;
1310     data->d_size = size;

1312     /*
1313      * There are several tags that are specific to the Solaris osabi
1314      * range which we unconditionally put into any dynamic section
1315      * we create (e.g. DT_SUNW_STRPAD or DT_SUNW_LDMACH). As such,

```

```

1316     * any Solaris object with a dynamic section should be tagged as
1317     * ELFOSABI_SOLARIS.
1318     */
1319     ofl->ofl_flags |= FLG_OF_OSABI;

1321     return ((uintptr_t)ofl->ofl_osdynamic);
1322 }

1324 /*
1325  * Build the GOT section and its associated relocation entries.
1326  */
1327 uintptr_t
1328 ld_make_got(Of1_desc *ofl)
1329 {
1330     Elf_Data    *data;
1331     Shdr        *shdr;
1332     Is_desc     *isec;
1333     size_t      size = (size_t)ofl->ofl_gotcnt * ld_targ.t_m.m_got_entsize;
1334     size_t      rsize = (size_t)ofl->ofl_relocotsz;

1336     if (new_section(ofl, SHT_PROGBITS, MSG_ORIG(MSG_SCN_GOT), 0,
1337                   &isec, &shdr, &data) == S_ERROR)
1338         return (S_ERROR);

1340     data->d_size = size;

1342     shdr->sh_flags |= SHF_WRITE;
1343     shdr->sh_size = (Xword)size;
1344     shdr->sh_entsize = ld_targ.t_m.m_got_entsize;

1346     ofl->ofl_osgot = ld_place_section(ofl, isec, NULL,
1347                                     ld_targ.t_id.id_got, NULL);
1348     if (ofl->ofl_osgot == (Os_desc *)S_ERROR)
1349         return (S_ERROR);

1351     ofl->ofl_osgot->os_szoutrels = (Xword)rsize;

1353     return (1);
1354 }

1356 /*
1357  * Build an interpreter section.
1358  */
1359 static uintptr_t
1360 make_interp(Of1_desc *ofl)
1361 {
1362     Shdr        *shdr;
1363     Elf_Data    *data;
1364     Is_desc     *isec;
1365     const char  *iname = ofl->ofl_interp;
1366     size_t      size;

1368     /*
1369     * If -z nointerp is in effect, don't create an interpreter section.
1370     */
1371     if (ofl->ofl_flags1 & FLG_OF1_NOINTRP)
1372         return (1);

1374     /*
1375     * An .interp section is always created for a dynamic executable.
1376     * A user can define the interpreter to use. This definition overrides
1377     * the default that would be recorded in an executable, and triggers
1378     * the creation of an .interp section in any other object. Presumably
1379     * the user knows what they are doing. Refer to the generic ELF ABI
1380     * section 5-4, and the ld(1) -I option.
1381     */

```

```

1382     if (((ofl->ofl_flags & (FLG_OF_DYNAMIC | FLG_OF_EXEC |
1383                          FLG_OF_RELOBJ)) != (FLG_OF_DYNAMIC | FLG_OF_EXEC)) && !iname)
1384         return (1);

1386     /*
1387     * In the case of a dynamic executable, supply a default interpreter
1388     * if the user has not specified their own.
1389     */
1390     if (iname == NULL)
1391         iname = ofl->ofl_interp = ld_targ.t_m.m_def_interp;

1393     size = strlen(iname) + 1;

1395     if (new_section(ofl, SHT_PROGBITS, MSG_ORIG(MSG_SCN_INTERP), 0,
1396                   &isec, &shdr, &data) == S_ERROR)
1397         return (S_ERROR);

1399     data->d_size = size;
1400     shdr->sh_size = (Xword)size;
1401     data->d_align = shdr->sh_addralign = 1;

1403     ofl->ofl_osinterp =
1404         ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_interp, NULL);
1405     return ((uintptr_t)ofl->ofl_osinterp);
1406 }

1408 /*
1409  * Common function used to build the SHT_SUNW_versym section, SHT_SUNW_syminfo
1410  * section, and SHT_SUNW_capinfo section. Each of these sections provide
1411  * additional symbol information, and their size parallels the associated
1412  * symbol table.
1413  */
1414 static Os_desc *
1415 make_sym_sec(Of1_desc *ofl, const char *sectname, Word stype, int ident)
1416 {
1417     Shdr        *shdr;
1418     Elf_Data    *data;
1419     Is_desc     *isec;

1421     /*
1422     * We don't know the size of this section yet, so set it to 0. The
1423     * size gets filled in after the associated symbol table is sized.
1424     */
1425     if (new_section(ofl, stype, sectname, 0, &isec, &shdr, &data) ==
1426         S_ERROR)
1427         return ((Os_desc *)S_ERROR);

1429     return (ld_place_section(ofl, isec, NULL, ident, NULL));
1430 }

1432 /*
1433  * Determine whether a symbol capability is redundant because the object
1434  * capabilities are more restrictive.
1435  */
1436 inline static int
1437 is_cap_redundant(Objcapset *ocapset, Objcapset *scapset)
1438 {
1439     Alist        *oalp, *salp;
1440     elfcap_mask_t  omsk, smsk;

1442     /*
1443     * Inspect any platform capabilities. If the object defines platform
1444     * capabilities, then the object will only be loaded for those
1445     * platforms. A symbol capability set that doesn't define the same
1446     * platforms is redundant, and a symbol capability that does not provide
1447     * at least one platform name that matches a platform name in the object

```

```

1448     * capabilities will never execute (as the object wouldn't have been
1449     * loaded).
1450     */
1451     oalp = ocapset->oc_plat.cl_val;
1452     salp = scapset->oc_plat.cl_val;
1453     if (oalp && ((salp == NULL) || cap_names_match(oalp, salp)))
1454         return (1);
1455
1456     /*
1457     * If the symbol capability set defines platforms, and the object
1458     * doesn't, then the symbol set is more restrictive.
1459     */
1460     if (salp && (oalp == NULL))
1461         return (0);
1462
1463     /*
1464     * Next, inspect any machine name capabilities. If the object defines
1465     * machine name capabilities, then the object will only be loaded for
1466     * those machines. A symbol capability set that doesn't define the same
1467     * machine names is redundant, and a symbol capability that does not
1468     * provide at least one machine name that matches a machine name in the
1469     * object capabilities will never execute (as the object wouldn't have
1470     * been loaded).
1471     */
1472     oalp = ocapset->oc_plat.cl_val;
1473     salp = scapset->oc_plat.cl_val;
1474     if (oalp && ((salp == NULL) || cap_names_match(oalp, salp)))
1475         return (1);
1476
1477     /*
1478     * If the symbol capability set defines machine names, and the object
1479     * doesn't, then the symbol set is more restrictive.
1480     */
1481     if (salp && (oalp == NULL))
1482         return (0);
1483
1484     /*
1485     * Next, inspect any hardware capabilities. If the objects hardware
1486     * capabilities are greater than or equal to that of the symbols
1487     * capabilities, then the symbol capability set is redundant. If the
1488     * symbols hardware capabilities are greater than the objects, then the
1489     * symbol set is more restrictive.
1490     *
1491     * Note that this is a somewhat arbitrary definition, as each capability
1492     * bit is independent of the others, and some of the higher order bits
1493     * could be considered to be less important than lower ones. However,
1494     * this is the only reasonable non-subjective definition.
1495     */
1496     omsk = ocapset->oc_hw_2.cm_val;
1497     smsk = scapset->oc_hw_2.cm_val;
1498     if ((omsk > smsk) || (omsk && (omsk == smsk)))
1499         return (1);
1500     if (omsk < smsk)
1501         return (0);
1502
1503     /*
1504     * Finally, inspect the remaining hardware capabilities.
1505     */
1506     omsk = ocapset->oc_hw_1.cm_val;
1507     smsk = scapset->oc_hw_1.cm_val;
1508     if ((omsk > smsk) || (omsk && (omsk == smsk)))
1509         return (1);
1510
1511     return (0);
1512 }

```

```

1514 /*
1515  * Capabilities values might have been assigned excluded values. These
1516  * excluded values should be removed before calculating any capabilities
1517  * sections size.
1518  */
1519 static void
1520 capmask_value(Lm_list *lml, Word type, Capmask *capmask, int *title)
1521 {
1522     /*
1523     * First determine whether any bits should be excluded.
1524     */
1525     if ((capmask->cm_val & capmask->cm_exc) == 0)
1526         return;
1527
1528     DBG_CALL(DBG_cap_post_title(lml, title));
1529
1530     DBG_CALL(DBG_cap_val_entry(lml, DBG_STATE_CURRENT, type,
1531                               capmask->cm_val, ld_targ.t_m.m_mach));
1532     DBG_CALL(DBG_cap_val_entry(lml, DBG_STATE_EXCLUDE, type,
1533                               capmask->cm_exc, ld_targ.t_m.m_mach));
1534
1535     capmask->cm_val &= ~capmask->cm_exc;
1536
1537     DBG_CALL(DBG_cap_val_entry(lml, DBG_STATE_RESOLVED, type,
1538                               capmask->cm_val, ld_targ.t_m.m_mach));
1539 }
1540
1541 static void
1542 capstr_value(Lm_list *lml, Word type, Caplist *caplist, int *title)
1543 {
1544     Aliste idx1, idx2;
1545     char *estr;
1546     Capstr *capstr;
1547     Boolean found = FALSE;
1548
1549     /*
1550     * First determine whether any strings should be excluded.
1551     */
1552     for (APLIST_TRAVERSE(caplist->cl_exc, idx1, estr)) {
1553         for (ALIST_TRAVERSE(caplist->cl_val, idx2, capstr)) {
1554             if (strcmp(estr, capstr->cs_str) == 0) {
1555                 found = TRUE;
1556                 break;
1557             }
1558         }
1559     }
1560
1561     if (found == FALSE)
1562         return;
1563
1564     /*
1565     * Traverse the current strings, then delete the excluded strings,
1566     * and finally display the resolved strings.
1567     */
1568     if (DBG_ENABLED) {
1569         Dbg_cap_post_title(lml, title);
1570         for (ALIST_TRAVERSE(caplist->cl_val, idx2, capstr)) {
1571             Dbg_cap_ptr_entry(lml, DBG_STATE_CURRENT, type,
1572                              capstr->cs_str);
1573         }
1574     }
1575     for (APLIST_TRAVERSE(caplist->cl_exc, idx1, estr)) {
1576         for (ALIST_TRAVERSE(caplist->cl_val, idx2, capstr)) {
1577             if (strcmp(estr, capstr->cs_str) == 0) {
1578                 DBG_CALL(DBG_cap_ptr_entry(lml,
1579                                           DBG_STATE_EXCLUDE, type, capstr->cs_str));
1580             }
1581         }
1582     }

```

```

1580         alist_delete(caplist->cl_val, &idx2);
1581         break;
1582     }
1583 }
1584 }
1585 if (DBG_ENABLED) {
1586     for (ALIST_TRAVERSE(caplist->cl_val, idx2, capstr)) {
1587         Dbg_cap_ptr_entry(lml, DBG_STATE_RESOLVED, type,
1588             capstr->cs_str);
1589     }
1590 }
1591 }

1593 /*
1594  * Build a capabilities section.
1595  */
1596 #define CAP_UPDATE(cap, capndx, tag, val) \
1597     cap->c_tag = tag; \
1598     cap->c_un.c_val = val; \
1599     cap++, capndx++;

1601 static uintptr_t
1602 make_cap(Of1_desc *of1, Word shtype, const char *shname, int ident)
1603 {
1604     Shdr      *shdr;
1605     Elf_Data  *data;
1606     Is_desc   *isec;
1607     Cap       *cap;
1608     size_t    size = 0;
1609     Word      capndx = 0;
1610     Str_tbl   *strtbl;
1611     Objcapset *ocapset = &of1->o1_ocapset;
1612     Aliste    idxl;
1613     Capstr    *capstr;
1614     int       title = 0;

1616     /*
1617      * Determine which string table to use for any CA_SUNW_MACH,
1618      * CA_SUNW_PLAT, or CA_SUNW_ID strings.
1619      */
1620     if (OFL_IS_STATIC_OBJ(of1))
1621         strtbl = of1->o1_strtab;
1622     else
1623         strtbl = of1->o1_dynstrtab;

1625     /*
1626      * If symbol capabilities have been requested, but none have been
1627      * created, warn the user. This scenario can occur if none of the
1628      * input relocatable objects defined any object capabilities.
1629      */
1630     if ((of1->o1_flags & FLG_OF_OTOSCAP) && (of1->o1_capsymcnt == 0))
1631         ld_eprintf(of1, ERR_WARNING, MSG_INTL(MSG_CAP_NOSYMSFOUND));

1633     /*
1634      * If symbol capabilities have been collected, but no symbols are left
1635      * referencing these capabilities, promote the capability groups back
1636      * to an object capability definition.
1637      */
1638     if ((of1->o1_flags & FLG_OF_OTOSCAP) && of1->o1_capsymcnt &&
1639         (of1->o1_capfamilies == NULL)) {
1640         ld_eprintf(of1, ERR_WARNING, MSG_INTL(MSG_CAP_NOSYMSFOUND));
1641         ld_cap_move_syntoobj(of1);
1642         of1->o1_capsymcnt = 0;
1643         of1->o1_capgroups = NULL;
1644         of1->o1_flags &= ~FLG_OF_OTOSCAP;
1645     }

```

```

1647     /*
1648      * Remove any excluded capabilities.
1649      */
1650     capstr_value(of1->o1_lml, CA_SUNW_PLAT, &ocapset->oc_plat, &title);
1651     capstr_value(of1->o1_lml, CA_SUNW_MACH, &ocapset->oc_mach, &title);
1652     capmask_value(of1->o1_lml, CA_SUNW_HW_2, &ocapset->oc_hw_2, &title);
1653     capmask_value(of1->o1_lml, CA_SUNW_HW_1, &ocapset->oc_hw_1, &title);
1654     capmask_value(of1->o1_lml, CA_SUNW_SF_1, &ocapset->oc_sf_1, &title);

1656     /*
1657      * Determine how many entries are required for any object capabilities.
1658      */
1659     size += alist_nitems(ocapset->oc_plat.cl_val);
1660     size += alist_nitems(ocapset->oc_mach.cl_val);
1661     if (ocapset->oc_hw_2.cm_val)
1662         size++;
1663     if (ocapset->oc_hw_1.cm_val)
1664         size++;
1665     if (ocapset->oc_sf_1.cm_val)
1666         size++;

1668     /*
1669      * Only identify a capabilities group if the group has content. If a
1670      * capabilities identifier exists, and no other capabilities have been
1671      * supplied, remove the identifier. This scenario could exist if a
1672      * user mistakenly defined a lone identifier, or if an identified group
1673      * was overridden so as to clear the existing capabilities and the
1674      * identifier was not also cleared.
1675      */
1676     if (ocapset->oc_id.cs_str) {
1677         if (size)
1678             size++;
1679         else
1680             ocapset->oc_id.cs_str = NULL;
1681     }
1682     if (size)
1683         size++; /* Add CA_SUNW_NULL */

1685     /*
1686      * Determine how many entries are required for any symbol capabilities.
1687      */
1688     if (of1->o1_capsymcnt) {
1689         /*
1690          * If there are no object capabilities, a CA_SUNW_NULL entry
1691          * is required before any symbol capabilities.
1692          */
1693         if (size == 0)
1694             size++;
1695         size += of1->o1_capsymcnt;
1696     }

1698     if (size == 0)
1699         return (NULL);

1701     if (new_section(of1, shtype, shname, size, &isec,
1702         &shdr, &data) == S_ERROR)
1703         return (S_ERROR);

1705     if ((data->d_buf = libld_malloc(shdr->sh_size)) == NULL)
1706         return (S_ERROR);

1708     cap = (Cap *)data->d_buf;

1710     /*
1711      * Fill in any object capabilities. If there is an identifier, then the

```

```

1712     * identifier comes first. The remaining items follow in precedence
1713     * order, although the order isn't important for runtime verification.
1714     */
1715     if (ocapset->oc_id.cs_str) {
1716         ofl->ofl_flags |= FLG_OF_CAPSTRS;
1717         if (st_insert(strtbl, ocapset->oc_id.cs_str) == -1)
1718             return (S_ERROR);
1719         ocapset->oc_id.cs_ndx = capndx;
1720         CAP_UPDATE(cap, capndx, CA_SUNW_ID, 0);
1721     }
1722     if (ocapset->oc_plat.cl_val) {
1723         ofl->ofl_flags |= (FLG_OF_PTCAP | FLG_OF_CAPSTRS);
1724
1725         /*
1726          * Insert any platform name strings in the appropriate string
1727          * table. The capability value can't be filled in yet, as the
1728          * final offset of the strings isn't known until later.
1729          */
1730         for (ALIST_TRAVERSE(ocapset->oc_plat.cl_val, idx1, capstr)) {
1731             if (st_insert(strtbl, capstr->cs_str) == -1)
1732                 return (S_ERROR);
1733             capstr->cs_ndx = capndx;
1734             CAP_UPDATE(cap, capndx, CA_SUNW_PLAT, 0);
1735         }
1736     }
1737     if (ocapset->oc_mach.cl_val) {
1738         ofl->ofl_flags |= (FLG_OF_PTCAP | FLG_OF_CAPSTRS);
1739
1740         /*
1741          * Insert the machine name strings in the appropriate string
1742          * table. The capability value can't be filled in yet, as the
1743          * final offset of the strings isn't known until later.
1744          */
1745         for (ALIST_TRAVERSE(ocapset->oc_mach.cl_val, idx1, capstr)) {
1746             if (st_insert(strtbl, capstr->cs_str) == -1)
1747                 return (S_ERROR);
1748             capstr->cs_ndx = capndx;
1749             CAP_UPDATE(cap, capndx, CA_SUNW_MACH, 0);
1750         }
1751     }
1752     if (ocapset->oc_hw_2.cm_val) {
1753         ofl->ofl_flags |= FLG_OF_PTCAP;
1754         CAP_UPDATE(cap, capndx, CA_SUNW_HW_2, ocapset->oc_hw_2.cm_val);
1755     }
1756     if (ocapset->oc_hw_1.cm_val) {
1757         ofl->ofl_flags |= FLG_OF_PTCAP;
1758         CAP_UPDATE(cap, capndx, CA_SUNW_HW_1, ocapset->oc_hw_1.cm_val);
1759     }
1760     if (ocapset->oc_sf_1.cm_val) {
1761         ofl->ofl_flags |= FLG_OF_PTCAP;
1762         CAP_UPDATE(cap, capndx, CA_SUNW_SF_1, ocapset->oc_sf_1.cm_val);
1763     }
1764     CAP_UPDATE(cap, capndx, CA_SUNW_NULL, 0);
1765
1766     /*
1767     * Fill in any symbol capabilities.
1768     */
1769     if (ofl->ofl_capgroups) {
1770         Cap_group      *cgp;
1771
1772         for (ALIST_TRAVERSE(ofl->ofl_capgroups, idx1, cgp)) {
1773             Objcapset  *scapset = &cgp->cg_set;
1774             Aliste      idx2;
1775             Is_desc     *isp;
1776
1777             cgp->cg_ndx = capndx;

```

```

1779         if (scapset->oc_id.cs_str) {
1780             ofl->ofl_flags |= FLG_OF_CAPSTRS;
1781             /*
1782              * Insert the identifier string in the
1783              * appropriate string table. The capability
1784              * value can't be filled in yet, as the final
1785              * offset of the string isn't known until later.
1786              */
1787             if (st_insert(strtbl,
1788                 scapset->oc_id.cs_str) == -1)
1789                 return (S_ERROR);
1790             scapset->oc_id.cs_ndx = capndx;
1791             CAP_UPDATE(cap, capndx, CA_SUNW_ID, 0);
1792         }
1793
1794         if (scapset->oc_plat.cl_val) {
1795             ofl->ofl_flags |= FLG_OF_CAPSTRS;
1796
1797             /*
1798              * Insert the platform name string in the
1799              * appropriate string table. The capability
1800              * value can't be filled in yet, as the final
1801              * offset of the string isn't known until later.
1802              */
1803             for (ALIST_TRAVERSE(scapset->oc_plat.cl_val,
1804                 idx2, capstr)) {
1805                 if (st_insert(strtbl,
1806                     capstr->cs_str) == -1)
1807                     return (S_ERROR);
1808                 capstr->cs_ndx = capndx;
1809                 CAP_UPDATE(cap, capndx,
1810                     CA_SUNW_PLAT, 0);
1811             }
1812         }
1813         if (scapset->oc_mach.cl_val) {
1814             ofl->ofl_flags |= FLG_OF_CAPSTRS;
1815
1816             /*
1817              * Insert the machine name string in the
1818              * appropriate string table. The capability
1819              * value can't be filled in yet, as the final
1820              * offset of the string isn't known until later.
1821              */
1822             for (ALIST_TRAVERSE(scapset->oc_mach.cl_val,
1823                 idx2, capstr)) {
1824                 if (st_insert(strtbl,
1825                     capstr->cs_str) == -1)
1826                     return (S_ERROR);
1827                 capstr->cs_ndx = capndx;
1828                 CAP_UPDATE(cap, capndx,
1829                     CA_SUNW_MACH, 0);
1830             }
1831         }
1832         if (scapset->oc_hw_2.cm_val) {
1833             CAP_UPDATE(cap, capndx, CA_SUNW_HW_2,
1834                 scapset->oc_hw_2.cm_val);
1835         }
1836         if (scapset->oc_hw_1.cm_val) {
1837             CAP_UPDATE(cap, capndx, CA_SUNW_HW_1,
1838                 scapset->oc_hw_1.cm_val);
1839         }
1840         if (scapset->oc_sf_1.cm_val) {
1841             CAP_UPDATE(cap, capndx, CA_SUNW_SF_1,
1842                 scapset->oc_sf_1.cm_val);
1843         }

```



```

1844     CAP_UPDATE(cap, capndx, CA_SUNW_NULL, 0);
1845
1846     /*
1847     * If any object capabilities are available, determine
1848     * whether these symbol capabilities are less
1849     * restrictive, and hence redundant.
1850     */
1851     if (((ofl->ofl_flags & FLG_OF_PTCAP) == 0) ||
1852         (is_cap_redundant(ocapset, scapset) == 0))
1853         continue;
1854
1855     /*
1856     * Indicate any files that provide redundant symbol
1857     * capabilities.
1858     */
1859     for (APLIST_TRAVERSE(cgp->cg_secs, idx2, isp) {
1860         ld_eprintf(ofl, ERR_WARNING,
1861             MSG_INTL(MSG_CAP_REDUNDANT),
1862             isp->is_file->ifl_name,
1863             EC_WORD(isp->is_scnndx), isp->is_name);
1864     }
1865 }
1866
1867 /*
1868 * If capabilities strings are required, the sh_info field of the
1869 * section header will be set to the associated string table.
1870 */
1871 if (ofl->ofl_flags & FLG_OF_CAPSTRS)
1872     shdr->sh_flags |= SHF_INFO_LINK;
1873
1874 /*
1875 * Place these capabilities in the output file.
1876 */
1877 if ((ofl->ofl_oscaps = ld_place_section(ofl, isec,
1878     NULL, ident, NULL)) == (Os_desc *)S_ERROR)
1879     return (S_ERROR);
1880
1881 /*
1882 * If symbol capabilities are required, then a .SUNW_capinfo section is
1883 * also created. This table will eventually be sized to match the
1884 * associated symbol table.
1885 */
1886 if (ofl->ofl_capfamilies) {
1887     if ((ofl->ofl_oscaps = make_sym_sec(ofl,
1888         MSG_ORIG(MSG_SCN_SUNWCAPINFO), SHT_SUNW_capinfo,
1889         ld_targ.t_id.id_capinfo)) == (Os_desc *)S_ERROR)
1890         return (S_ERROR);
1891 }
1892
1893 /*
1894 * If we're generating a dynamic object, capabilities family
1895 * members are maintained in a .SUNW_capchain section.
1896 */
1897 if (ofl->ofl_capchaincnt &&
1898     ((ofl->ofl_flags & FLG_OF_RELOBJ) == 0)) {
1899     if (new_section(ofl, SHT_SUNW_capchain,
1900         MSG_ORIG(MSG_SCN_SUNWCAPCHAIN),
1901         ofl->ofl_capchaincnt, &isec, &shdr,
1902         &data) == S_ERROR)
1903         return (S_ERROR);
1904
1905     ofl->ofl_oscapschain = ld_place_section(ofl, isec,
1906         NULL, ld_targ.t_id.id_capchain, NULL);
1907     if (ofl->ofl_oscapschain == (Os_desc *)S_ERROR)
1908         return (S_ERROR);

```

```

1910     }
1911 }
1912 return (1);
1913 }
1914 #undef CAP_UPDATE
1915
1916 /*
1917 * Build the PLT section and its associated relocation entries.
1918 */
1919 static uintptr_t
1920 make_plt(Of1_desc *ofl)
1921 {
1922     Shdr      *shdr;
1923     Elf_Data  *data;
1924     Is_desc   *isec;
1925     size_t    size = ld_targ.t_m.m_plt_reservsz +
1926         (((size_t)ofl->ofl_pltcnt + (size_t)ofl->ofl_pltpad) *
1927         ld_targ.t_m.m_plt_entsize);
1928     size_t    rsize = (size_t)ofl->ofl_relocpltsz;
1929
1930     /*
1931     * On sparc, account for the NOP at the end of the plt.
1932     */
1933     if (ld_targ.t_m.m_mach == LD_TARG_BYCLASS(EM_SPARC, EM_SPARCV9))
1934         size += sizeof (Word);
1935
1936     if (new_section(ofl, SHT_PROGBITS, MSG_ORIG(MSG_SCN_PLT), 0,
1937         &isec, &shdr, &data) == S_ERROR)
1938         return (S_ERROR);
1939
1940     data->d_size = size;
1941     data->d_align = ld_targ.t_m.m_plt_align;
1942
1943     shdr->sh_flags = ld_targ.t_m.m_plt_shf_flags;
1944     shdr->sh_size = (Xword)size;
1945     shdr->sh_addralign = ld_targ.t_m.m_plt_align;
1946     shdr->sh_entsize = ld_targ.t_m.m_plt_entsize;
1947
1948     ofl->ofl_osplt = ld_place_section(ofl, isec, NULL,
1949         ld_targ.t_id.id_plt, NULL);
1950     if (ofl->ofl_osplt == (Os_desc *)S_ERROR)
1951         return (S_ERROR);
1952
1953     ofl->ofl_osplt->os_szoutrels = (Xword)rsize;
1954
1955     return (1);
1956 }
1957
1958 /*
1959 * Make the hash table. Only built for dynamic executables and shared
1960 * libraries, and provides hashed lookup into the global symbol table
1961 * (.dynsym) for the run-time linker to resolve symbol lookups.
1962 */
1963 static uintptr_t
1964 make_hash(Of1_desc *ofl)
1965 {
1966     Shdr      *shdr;
1967     Elf_Data  *data;
1968     Is_desc   *isec;
1969     size_t    size;
1970     Word      nsyms = ofl->ofl_globcnt;
1971     size_t    cnt;
1972
1973     /*
1974     * Allocate section header structures. We set entcnt to 0
1975     * because it's going to change after we place this section.

```

```

1976  */
1977  if (new_section(ofl, SHT_HASH, MSG_ORIG(MSG_SCN_HASH), 0,
1978      &isec, &shdr, &data) == S_ERROR)
1979      return (S_ERROR);

1981  /*
1982  * Place the section first since it will affect the local symbol
1983  * count.
1984  */
1985  ofl->ofl_oshash =
1986      ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_hash, NULL);
1987  if (ofl->ofl_oshash == (Os_desc *)S_ERROR)
1988      return (S_ERROR);

1990  /*
1991  * Calculate the number of output hash buckets.
1992  */
1993  ofl->ofl_hashbkts = findprime(nsyms);

1995  /*
1996  * The size of the hash table is determined by
1997  *
1998  *   i.    the initial nbucket and nchain entries (2)
1999  *   ii.   the number of buckets (calculated above)
2000  *   iii.  the number of chains (this is based on the number of
2001  *         symbols in the .dynsym array).
2002  */
2003  cnt = 2 + ofl->ofl_hashbkts + DYNYSM_ALL_CNT(ofl);
2004  size = cnt * shdr->sh_entsize;

2006  /*
2007  * Finalize the section header and data buffer initialization.
2008  */
2009  if ((data->d_buf = libld_calloc(size, 1)) == NULL)
2010      return (S_ERROR);
2011  data->d_size = size;
2012  shdr->sh_size = (Xword)size;

2014  return (1);
2015 }

2017 /*
2018 * Generate the standard symbol table. Contains all locals and globals,
2019 * and resides in a non-allocatable section (ie. it can be stripped).
2020 */
2021 static uintptr_t
2022 make_syntab(Of1_desc *ofl)
2023 {
2024     Shdr      *shdr;
2025     Elf_Data  *data;
2026     Is_desc   *isec;
2027     Is_desc   *xisec = 0;
2028     size_t    size;
2029     Word      symcnt;

2031  /*
2032  * Create the section headers. Note that we supply an ent_cnt
2033  * of 0. We won't know the count until the section has been placed.
2034  */
2035  if (new_section(ofl, SHT_SYMTAB, MSG_ORIG(MSG_SCN_SYMTAB), 0,
2036      &isec, &shdr, &data) == S_ERROR)
2037      return (S_ERROR);

2039  /*
2040  * Place the section first since it will affect the local symbol
2041  * count.

```

```

2042  */
2043  if ((ofl->ofl_ossymtab = ld_place_section(ofl, isec, NULL,
2044      ld_targ.t_id.id_syntab, NULL)) == (Os_desc *)S_ERROR)
2045      return (S_ERROR);

2047  /*
2048  * At this point we've created all but the 'shstrtab' section.
2049  * Determine if we have to use 'Extended Sections'. If so - then
2050  * also create a SHT_SYMTAB_SHNDX section.
2051  */
2052  if ((ofl->ofl_shdrcont + 1) >= SHN_LORESERVE) {
2053      Shdr      *xshdr;
2054      Elf_Data  *xdata;

2056      if (new_section(ofl, SHT_SYMTAB_SHNDX,
2057          MSG_ORIG(MSG_SCN_SYMTAB_SHNDX), 0, &xisec,
2058          &xshdr, &xdata) == S_ERROR)
2059          return (S_ERROR);

2061      if ((ofl->ofl_ossymshndx = ld_place_section(ofl, xisec, NULL,
2062          ld_targ.t_id.id_syntab_ndx, NULL)) == (Os_desc *)S_ERROR)
2063          return (S_ERROR);
2064  }

2066  /*
2067  * Calculated number of symbols, which need to be augmented by
2068  * the (yet to be created) .shstrtab entry.
2069  */
2070  symcnt = (size_t)(1 + SYMTAB_ALL_CNT(ofl));
2071  size = symcnt * shdr->sh_entsize;

2073  /*
2074  * Finalize the section header and data buffer initialization.
2075  */
2076  data->d_size = size;
2077  shdr->sh_size = (Xword)size;

2079  /*
2080  * If we created a SHT_SYMTAB_SHNDX - then set it's sizes too.
2081  */
2082  if (xisec) {
2083      size_t  xsize = symcnt * sizeof (Word);

2085      xisec->is_indata->d_size = xsize;
2086      xisec->is_shdr->sh_size = (Xword)xsize;
2087  }

2089  return (1);
2090 }

2092 /*
2093 * Build a dynamic symbol table. These tables reside in the text
2094 * segment of a dynamic executable or shared library.
2095 *
2096 *   .SUNW_ldynsym contains local function symbols
2097 *   .dynsym contains only globals symbols
2098 *
2099 * The two tables are created adjacent to each other, with .SUNW_ldynsym
2100 * coming first.
2101 */
2102 static uintptr_t
2103 make_dynsym(Of1_desc *ofl)
2104 {
2105     Shdr      *shdr, *lshdr;
2106     Elf_Data  *data, *ldata;
2107     Is_desc   *isec, *lisec;

```

```

2108     size_t      size;
2109     Xword      cnt;
2110     int        allow_ldynsym;

2112 /*
2113  * Unless explicitly disabled, always produce a .SUNW_ldynsym section
2114  * when it is allowed by the file type, even if the resulting
2115  * table only ends up with a single STT_FILE in it. There are
2116  * two reasons: (1) It causes the generation of the DT_SUNW_SYMTAB
2117  * entry in the .dynamic section, which is something we would
2118  * like to encourage, and (2) Without it, we cannot generate
2119  * the associated .SUNW_dyn[sym|tls]sort sections, which are of
2120  * value to DTrace.
2121  *
2122  * In practice, it is extremely rare for an object not to have
2123  * local symbols for .SUNW_ldynsym, so 99% of the time, we'd be
2124  * doing it anyway.
2125  */
2126 allow_ldynsym = OFL_ALLOW_LDYSYM(ofl);

2128 /*
2129  * Create the section headers. Note that we supply an ent_cnt
2130  * of 0. We won't know the count until the section has been placed.
2131  */
2132 if (allow_ldynsym && new_section(ofl, SHT_SUNW_LDYSYM,
2133     MSG_ORIG(MSG_SCN_LDYSYM), 0, &lisec, &lshdr, &ldata) == S_ERROR)
2134     return (S_ERROR);

2136 if (new_section(ofl, SHT_DYNSYM, MSG_ORIG(MSG_SCN_DYNSYM), 0,
2137     &lisec, &lshdr, &ldata) == S_ERROR)
2138     return (S_ERROR);

2140 /*
2141  * Place the section(s) first since it will affect the local symbol
2142  * count.
2143  */
2144 if (allow_ldynsym &&
2145     ((ofl->ofl_osldynsym = ld_place_section(ofl, lisec, NULL,
2146     ld_targ.t_id.id_ldynsym, NULL)) == (Os_desc *)S_ERROR))
2147     return (S_ERROR);
2148 ofl->ofl_osdynsym =
2149     ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_dynsym, NULL);
2150 if (ofl->ofl_osdynsym == (Os_desc *)S_ERROR)
2151     return (S_ERROR);

2153 cnt = DYNSYM_ALL_CNT(ofl);
2154 size = (size_t)cnt * shdr->sh_entsize;

2156 /*
2157  * Finalize the section header and data buffer initialization.
2158  */
2159 data->d_size = size;
2160 shdr->sh_size = (Xword)size;

2162 /*
2163  * An ldynsym contains local function symbols. It is not
2164  * used for linking, but if present, serves to allow better
2165  * stack traces to be generated in contexts where the symtab
2166  * is not available. (dladdr(), or stripped executable/library files).
2167  */
2168 if (allow_ldynsym) {
2169     cnt = 1 + ofl->ofl_dynlocscnt + ofl->ofl_dynscopecnt;
2170     size = (size_t)cnt * shdr->sh_entsize;

2172     ldata->d_size = size;
2173     lshdr->sh_size = (Xword)size;

```

```

2174     }

2176     return (1);
2177 }

2179 /*
2180  * Build .SUNW_dynsym sort and/or .SUNW_dyntlssort sections. These are
2181  * index sections for the .SUNW_ldynsym/.dynsym pair that present data
2182  * and function symbols sorted by address.
2183  */
2184 static uintptr_t
2185 make_dynsort(Ofl_desc *ofl)
2186 {
2187     Shdr      *shdr;
2188     Elf_Data  *data;
2189     Is_desc   *isec;

2191     /* Only do it if the .SUNW_ldynsym section is present */
2192     if (!OFL_ALLOW_LDYSYM(ofl))
2193         return (1);

2195     /* .SUNW_dynsym sort */
2196     if (ofl->ofl_dynsym sortcnt > 0) {
2197         if (new_section(ofl, SHT_SUNW_SYMSORT,
2198             MSG_ORIG(MSG_SCN_DYNSYMSORT), ofl->ofl_dynsym sortcnt,
2199             &isec, &lshdr, &ldata) == S_ERROR)
2200             return (S_ERROR);

2202         if ((ofl->ofl_osdynsym sort = ld_place_section(ofl, isec, NULL,
2203             ld_targ.t_id.id_dynsym sort, NULL)) == (Os_desc *)S_ERROR)
2204             return (S_ERROR);
2205     }

2207     /* .SUNW_dyntlssort */
2208     if (ofl->ofl_dyntlssortcnt > 0) {
2209         if (new_section(ofl, SHT_SUNW_TLSSORT,
2210             MSG_ORIG(MSG_SCN_DYNTLSSORT),
2211             ofl->ofl_dyntlssortcnt, &isec, &lshdr, &ldata) == S_ERROR)
2212             return (S_ERROR);

2214         if ((ofl->ofl_osdyntlssort = ld_place_section(ofl, isec, NULL,
2215             ld_targ.t_id.id_dynsym sort, NULL)) == (Os_desc *)S_ERROR)
2216             return (S_ERROR);
2217     }

2219     return (1);
2220 }

2222 /*
2223  * Helper routine for make_dynsym_shndx. Builds a
2224  * a SHT_SYMTAB_SHNDX for .dynsym or .SUNW_ldynsym, without knowing
2225  * which one it is.
2226  */
2227 static uintptr_t
2228 make_dyn_shndx(Ofl_desc *ofl, const char *shname, Os_desc *symtab,
2229     Os_desc **ret_os)
2230 {
2231     Is_desc   *isec;
2232     Is_desc   *dynsymisec;
2233     Shdr      *shdr;
2234     Elf_Data  *data;

2236     dynsymisec = ld_os_first_isdesc(symtab);
2237     dynshdr = dynsymisec->is_shdr;

2239     if (new_section(ofl, SHT_SYMTAB_SHNDX, shname,

```

```

2240     (dynshdr->sh_size / dynshdr->sh_entsize),
2241     &isec, &shdr, &data) == S_ERROR)
2242     return (S_ERROR);

2244     if ((*ret_os = ld_place_section(ofl, isec, NULL,
2245     ld_targ.t_id.id_dynsym_ndx, NULL)) == (Os_desc *)S_ERROR)
2246     return (S_ERROR);

2248     assert(*ret_os);

2250     return (1);
2251 }

2253 /*
2254  * Build a SHT_SYMTAB_SHNDX for the .dynsym, and .SUNW_ldynsym
2255  */
2256 static uintptr_t
2257 make_dynsym_shndx(Of1_desc *ofl)
2258 {
2259     /*
2260     * If there is a .SUNW_ldynsym, generate a section for its extended
2261     * index section as well.
2262     */
2263     if (OFL_ALLOW_LDYNsym(ofl)) {
2264         if (make_dyn_shndx(ofl, MSG_ORIG(MSG_SCN_LDYNsym_SHNDX),
2265         ofl->o1_osldynsym, &o1->o1_osldynshndx) == S_ERROR)
2266             return (S_ERROR);
2267     }

2269     /* The Generate a section for the dynsym */
2270     if (make_dyn_shndx(ofl, MSG_ORIG(MSG_SCN_DYNsym_SHNDX),
2271     ofl->o1_osdynsym, &o1->o1_osdynshndx) == S_ERROR)
2272     return (S_ERROR);

2274     return (1);
2275 }

2278 /*
2279  * Build a string table for the section headers.
2280  */
2281 static uintptr_t
2282 make_shstrtab(Of1_desc *ofl)
2283 {
2284     Shdr      *shdr;
2285     Elf_Data  *data;
2286     Is_desc   *isec;
2287     size_t    size;

2289     if (new_section(ofl, SHT_STRTAB, MSG_ORIG(MSG_SCN_SHSTRTAB),
2290     0, &isec, &shdr, &data) == S_ERROR)
2291     return (S_ERROR);

2293     /*
2294     * Place the section first, as it may effect the number of section
2295     * headers to account for.
2296     */
2297     ofl->o1_osshstrtab =
2298     ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_note, NULL);
2299     if (ofl->o1_osshstrtab == (Os_desc *)S_ERROR)
2300     return (S_ERROR);

2302     size = st_getstrtab_sz(ofl->o1_shdrsttab);
2303     assert(size > 0);

2305     data->d_size = size;

```

```

2306     shdr->sh_size = (Xword)size;

2308     return (1);
2309 }

2311 /*
2312  * Build a string section for the standard symbol table.
2313  */
2314 static uintptr_t
2315 make_strtab(Of1_desc *ofl)
2316 {
2317     Shdr      *shdr;
2318     Elf_Data  *data;
2319     Is_desc   *isec;
2320     size_t    size;

2322     /*
2323     * This string table consists of all the global and local symbols.
2324     * Account for null bytes at end of the file name and the beginning
2325     * of section.
2326     */
2327     if (st_insert(ofl->o1_strtab, ofl->o1_name) == -1)
2328     return (S_ERROR);

2330     size = st_getstrtab_sz(ofl->o1_strtab);
2331     assert(size > 0);

2333     if (new_section(ofl, SHT_STRTAB, MSG_ORIG(MSG_SCN_STRTAB),
2334     0, &isec, &shdr, &data) == S_ERROR)
2335     return (S_ERROR);

2337     /* Set the size of the data area */
2338     data->d_size = size;
2339     shdr->sh_size = (Xword)size;

2341     ofl->o1_osstrtab =
2342     ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_strtab, NULL);
2343     return ((uintptr_t)ofl->o1_osstrtab);
2344 }

2346 /*
2347  * Build a string table for the dynamic symbol table.
2348  */
2349 static uintptr_t
2350 make_dynstr(Of1_desc *ofl)
2351 {
2352     Shdr      *shdr;
2353     Elf_Data  *data;
2354     Is_desc   *isec;
2355     size_t    size;

2357     /*
2358     * If producing a .SUNW_ldynsym, account for the initial STT_FILE
2359     * symbol that precedes the scope reduced global symbols.
2360     */
2361     if (OFL_ALLOW_LDYNsym(ofl)) {
2362         if (st_insert(ofl->o1_dynstrtab, ofl->o1_name) == -1)
2363             return (S_ERROR);
2364         ofl->o1_dynscopecnt++;
2365     }

2367     /*
2368     * Account for any local, named register symbols. These locals are
2369     * required for reference from DT_REGISTER .dynamic entries.
2370     */
2371     if (ofl->o1_regysms) {

```

```

2372     int     ndx;
2374     for (ndx = 0; ndx < ofl->ofl_regysmsno; ndx++) {
2375         Sym_desc *sdp;
2377         if ((sdp = ofl->ofl_regysms[ndx]) == NULL)
2378             continue;
2380         if (!SYM_IS_HIDDEN(sdp) &&
2381             (ELF_ST_BIND(sdp->sd_sym->st_info) != STB_LOCAL))
2382             continue;
2384         if (sdp->sd_sym->st_name == NULL)
2385             continue;
2387         if (st_insert(ofl->ofl_dynstrtab, sdp->sd_name) == -1)
2388             return (S_ERROR);
2389     }
2390 }
2392 /*
2393  * Reserve entries for any per-symbol auxiliary/filter strings.
2394  */
2395 if (ofl->ofl_dtsfltrs != NULL) {
2396     Dfltr_desc *dftp;
2397     Aliste     idx;
2399     for (ALIST_TRAVERSE(ofl->ofl_dtsfltrs, idx, dftp))
2400         if (st_insert(ofl->ofl_dynstrtab, dftp->dft_str) == -1)
2401             return (S_ERROR);
2402 }
2404 size = st_getstrtab_sz(ofl->ofl_dynstrtab);
2405 assert(size > 0);
2407 if (new_section(ofl, SHT_STRTAB, MSG_ORIG(MSG_SCN_DYNSTR),
2408                0, &isec, &shdr, &data) == S_ERROR)
2409     return (S_ERROR);
2411 /* Make it allocable if necessary */
2412 if (!(ofl->ofl_flags & FLG_OF_RELOBJ))
2413     shdr->sh_flags |= SHF_ALLOC;
2415 /* Set the size of the data area */
2416 data->d_size = size + DYNSTR_EXTRA_PAD;
2418 shdr->sh_size = (Xword)size;
2420 ofl->ofl_osdynstr =
2421     ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_dynstr, NULL);
2422 return ((uintptr_t)ofl->ofl_osdynstr);
2423 }
2425 /*
2426  * Generate an output relocation section which will contain the relocation
2427  * information to be applied to the 'osp' section.
2428  *
2429  * If (osp == NULL) then we are creating the coalesced relocation section
2430  * for an executable and/or a shared object.
2431  */
2432 static uintptr_t
2433 make_reloc(Of1_desc *ofl, Os_desc *osp)
2434 {
2435     Shdr     *shdr;
2436     Elf_Data *data;
2437     Is_desc  *isec;

```

```

2438     size_t     size;
2439     Xword      sh_flags;
2440     char       *sectname;
2441     Os_desc    *rosp;
2442     Word       relsize;
2443     const char *rel_prefix;
2445     /* LINTED */
2446     if (ld_targ.t_m.m_rel_sht_type == SHT_REL) {
2447         /* REL */
2448         relsize = sizeof (Rel);
2449         rel_prefix = MSG_ORIG(MSG_SCN_REL);
2450     } else {
2451         /* RELA */
2452         relsize = sizeof (Rela);
2453         rel_prefix = MSG_ORIG(MSG_SCN_RELA);
2454     }
2456     if (osp) {
2457         size = osp->os_szoutrels;
2458         sh_flags = osp->os_shdr->sh_flags;
2459         if ((sectname = libld_malloc(strlen(rel_prefix) +
2460                                     strlen(osp->os_name) + 1)) == 0)
2461             return (S_ERROR);
2462         (void) strcpy(sectname, rel_prefix);
2463         (void) strcat(sectname, osp->os_name);
2464     } else if (ofl->ofl_flags & FLG_OF_COMREL) {
2465         size = (ofl->ofl_relocct - ofl->ofl_relocctsub) * relsize;
2466         sh_flags = SHF_ALLOC;
2467         sectname = (char *)MSG_ORIG(MSG_SCN_SUNWRELOC);
2468     } else {
2469         size = ofl->ofl_relocrelsz;
2470         sh_flags = SHF_ALLOC;
2471         sectname = (char *)rel_prefix;
2472     }
2474     /*
2475      * Keep track of total size of 'output relocations' (to be stored
2476      * in .dynamic)
2477      */
2478     /* LINTED */
2479     ofl->ofl_relocsz += (Xword)size;
2481     if (new_section(ofl, ld_targ.t_m.m_rel_sht_type, sectname, 0, &isec,
2482                    &shdr, &data) == S_ERROR)
2483         return (S_ERROR);
2485     data->d_size = size;
2487     shdr->sh_size = (Xword)size;
2488     if (OFL_ALLOW_DYNSYM(ofl) && (sh_flags & SHF_ALLOC))
2489         shdr->sh_flags = SHF_ALLOC;
2491     if (osp) {
2492         /*
2493          * The sh_info field of the SHT_REL* sections points to the
2494          * section the relocations are to be applied to.
2495          */
2496         shdr->sh_flags |= SHF_INFO_LINK;
2497     }
2499     rosp = ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_rel, NULL);
2500     if (rosp == (Os_desc *)S_ERROR)
2501         return (S_ERROR);
2503     /*

```

```

2504     * Associate this relocation section to the section its going to
2505     * relocate.
2506     */
2507     if (osp) {
2508         Aliste idx;
2509         Is_desc *risp;
2510
2511         /*
2512          * This is used primarily so that we can update
2513          * SHT_GROUP[sect_no] entries to point to the
2514          * created output relocation sections.
2515          */
2516         for (APLIST_TRAVERSE(osp->os_relisdescs, idx, risp)) {
2517             risp->is_osdesc = rosp;
2518
2519             /*
2520              * If the input relocation section had the SHF_GROUP
2521              * flag set - propagate it to the output relocation
2522              * section.
2523              */
2524             if (risp->is_shdr->sh_flags & SHF_GROUP) {
2525                 rosp->os_shdr->sh_flags |= SHF_GROUP;
2526                 break;
2527             }
2528             osp->os_relosdesc = rosp;
2529         } else
2530             ofl->ofl_osrel = rosp;
2531
2532     /*
2533      * If this is the first relocation section we've encountered save it
2534      * so that the .dynamic entry can be initialized accordingly.
2535      */
2536     if (ofl->ofl_osrelhead == (Os_desc *)0)
2537         ofl->ofl_osrelhead = rosp;
2538
2539     return (1);
2540 }
2541
2542 /*
2543  * Generate version needed section.
2544  */
2545 static uintptr_t
2546 make_verneed(Of1_desc *of1)
2547 {
2548     Shdr      *shdr;
2549     Elf_Data  *data;
2550     Is_desc   *isec;
2551
2552     /*
2553      * verneed sections do not have a constant element size, so the
2554      * value of ent_cnt specified here (0) is meaningless.
2555      */
2556     if (new_section(of1, SHT_SUNW_verneed, MSG_ORIG(MSG_SCN_SUNWVERSION),
2557         0, &isec, &shdr, &data) == S_ERROR)
2558         return (S_ERROR);
2559
2560     /* During version processing we calculated the total size. */
2561     data->d_size = of1->ofl_verneedsz;
2562     shdr->sh_size = (Xword)ofl->ofl_verneedsz;
2563
2564     ofl->ofl_osverneed =
2565         ld_place_section(of1, isec, NULL, ld_targ.t_id.id_version, NULL);
2566     return ((uintptr_t)ofl->ofl_osverneed);
2567 }
2568 }

```

```

2570 /*
2571  * Generate a version definition section.
2572  */
2573  * o the SHT_SUNW_verdef section defines the versions that exist within this
2574  * image.
2575  */
2576 static uintptr_t
2577 make_verdef(Of1_desc *of1)
2578 {
2579     Shdr      *shdr;
2580     Elf_Data  *data;
2581     Is_desc   *isec;
2582     Ver_desc  *vdp;
2583     Str_tbl   *strtab;
2584
2585     /*
2586      * Reserve a string table entry for the base version dependency (other
2587      * dependencies have symbol representations, which will already be
2588      * accounted for during symbol processing).
2589      */
2590     vdp = (Ver_desc *)ofl->ofl_verdesc->apl_data[0];
2591
2592     if (OFL_IS_STATIC_OBJ(of1))
2593         strtab = ofl->ofl_strtab;
2594     else
2595         strtab = ofl->ofl_dynstrtab;
2596
2597     if (st_insert(strtab, vdp->vd_name) == -1)
2598         return (S_ERROR);
2599
2600     /*
2601      * verdef sections do not have a constant element size, so the
2602      * value of ent_cnt specified here (0) is meaningless.
2603      */
2604     if (new_section(of1, SHT_SUNW_verdef, MSG_ORIG(MSG_SCN_SUNWVERSION),
2605         0, &isec, &shdr, &data) == S_ERROR)
2606         return (S_ERROR);
2607
2608     /* During version processing we calculated the total size. */
2609     data->d_size = ofl->ofl_verdefsz;
2610     shdr->sh_size = (Xword)ofl->ofl_verdefsz;
2611
2612     ofl->ofl_osverdef =
2613         ld_place_section(of1, isec, NULL, ld_targ.t_id.id_version, NULL);
2614     return ((uintptr_t)ofl->ofl_osverdef);
2615 }
2616
2617 /*
2618  * This routine is called when -z nopartial is in effect.
2619  */
2620 uintptr_t
2621 ld_make_parexp_data(Of1_desc *of1, size_t size, Xword align)
2622 {
2623     Shdr      *shdr;
2624     Elf_Data  *data;
2625     Is_desc   *isec;
2626     *osp;
2627
2628     if (new_section(of1, SHT_PROGBITS, MSG_ORIG(MSG_SCN_DATA), 0,
2629         &isec, &shdr, &data) == S_ERROR)
2630         return (S_ERROR);
2631
2632     shdr->sh_flags |= SHF_WRITE;
2633     data->d_size = size;
2634     shdr->sh_size = (Xword)size;
2635     if (align != 0) {

```

```

2636         data->d_align = align;
2637         shdr->sh_addralign = align;
2638     }

2640     if ((data->d_buf = libld_calloc(size, 1)) == NULL)
2641         return (S_ERROR);

2643     /*
2644     * Retain handle to this .data input section. Variables using move
2645     * sections (partial initialization) will be redirected here when
2646     * such global references are added and '-z nopartial' is in effect.
2647     */
2648     ofl->ofl_isparexpn = isec;
2649     osp = ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_data, NULL);
2650     if (osp == (Os_desc *)S_ERROR)
2651         return (S_ERROR);

2653     if (!(osp->os_flags & FLG_OS_OUTREL)) {
2654         ofl->ofl_dynshdrctnt++;
2655         osp->os_flags |= FLG_OS_OUTREL;
2656     }
2657     return (1);
2658 }

2660 /*
2661 * Make .sunwmove section
2662 */
2663 uintptr_t
2664 ld_make_sunwmove(Of1_desc *ofl, int mv_nums)
2665 {
2666     Shdr      *shdr;
2667     Elf_Data  *data;
2668     Is_desc   *isec;
2669     Aliste    idx;
2670     Sym_desc  *sdp;
2671     int       cnt = 1;

2674     if (new_section(ofl, SHT_SUNW_move, MSG_ORIG(MSG_SCN_SUNWMOVE),
2675         mv_nums, &isec, &shdr, &data) == S_ERROR)
2676         return (S_ERROR);

2678     if ((data->d_buf = libld_calloc(data->d_size, 1)) == NULL)
2679         return (S_ERROR);

2681     /*
2682     * Copy move entries
2683     */
2684     for (APLIST_TRAVERSE(ofl->ofl_parsyms, idx, sdp)) {
2685         Aliste    idx2;
2686         Mv_desc    *mdp;

2688         if (sdp->sd_flags & FLG_SY_PAREXPON)
2689             continue;

2691         for (ALIST_TRAVERSE(sdp->sd_move, idx2, mdp))
2692             mdp->md_oidx = cnt++;
2693     }

2695     if ((ofl->ofl_osmove = ld_place_section(ofl, isec, NULL, 0, NULL)) ==
2696         (Os_desc *)S_ERROR)
2697         return (S_ERROR);

2699     return (1);
2700 }

```

```

2702 /*
2703 * Given a relocation descriptor that references a string table
2704 * input section, locate the string referenced and return a pointer
2705 * to it.
2706 */
2707 static const char *
2708 strmerge_get_reloc_str(Of1_desc *ofl, Rel_desc *rsp)
2709 {
2710     Sym_desc *sdp = rsp->rel_sym;
2711     Xword    str_off;

2713     /*
2714     * In the case of an STT_SECTION symbol, the addend of the
2715     * relocation gives the offset into the string section. For
2716     * other symbol types, the symbol value is the offset.
2717     */

2719     if (ELF_ST_TYPE(sdp->sd_sym->st_info) != STT_SECTION) {
2720         str_off = sdp->sd_sym->st_value;
2721     } else if ((rsp->rel_flags & FLG_REL_RELA) == FLG_REL_RELA) {
2722         /*
2723         * For SHT_RELA, the addend value is found in the
2724         * rel_raddend field of the relocation.
2725         */
2726         str_off = rsp->rel_raddend;
2727     } else { /* REL and STT_SECTION */
2728         /*
2729         * For SHT_REL, the "addend" is not part of the relocation
2730         * record. Instead, it is found at the relocation target
2731         * address.
2732         */
2733         uchar_t *addr = (uchar_t *)((uintptr_t)rsp->rel_roffset +
2734             (uintptr_t)rsp->rel_isdesc->is_indata->d_buf);

2736         if (ld_reloc_targval_get(ofl, rsp, addr, &str_off) == 0)
2737             return (0);
2738     }

2740     return (str_off + (char *)sdp->sd_isc->is_indata->d_buf);
2741 }

2743 /*
2744 * First pass over the relocation records for string table merging.
2745 * Build lists of relocations and symbols that will need modification,
2746 * and insert the strings they reference into the mstrtab string table.
2747 *
2748 * entry:
2749 * ofl, osp - As passed to ld_make_strmerge().
2750 * mstrtab - String table to receive input strings. This table
2751 * must be in its first (initialization) pass and not
2752 * yet cooked (st_getstrtab_sz() not yet called).
2753 * rel_alpp - APlist to receive pointer to any relocation
2754 * descriptors with STT_SECTION symbols that reference
2755 * one of the input sections being merged.
2756 * sym_alpp - APlist to receive pointer to any symbols that reference
2757 * one of the input sections being merged.
2758 * rcp - Pointer to cache of relocation descriptors to examine.
2759 * Either &ofl->ofl_actrels (active relocations)
2760 * or &ofl->ofl_outrels (output relocations).
2761 *
2762 * exit:
2763 * On success, rel_alpp and sym_alpp are updated, and
2764 * any strings in the mergeable input sections referenced by
2765 * a relocation has been entered into mstrtab. True (1) is returned.
2766 *
2767 * On failure, False (0) is returned.

```

```

2768 */
2769 static int
2770 strmerge_pass1(Of1_desc *of1, Os_desc *osp, Str_tbl *mstrtab,
2771             APlist **rel_alpp, APlist **sym_alpp, Rel_cache *rcp)
2772 {
2773     Aliste      idx;
2774     Rel_cachebuf *rcbp;
2775     Sym_desc    *sdp;
2776     Sym_desc    *last_sdp = NULL;
2777     Rel_desc    *rsp;
2778     const char  *name;

2780     REL_CACHE_TRAVERSE(rcp, idx, rcbp, rsp) {
2781         sdp = rsp->rel_sym;
2782         if ((sdp->sd_isc == NULL) || ((sdp->sd_isc->is_flags &
2783             (FLG_IS_DISCARD | FLG_IS_INSTRMRG)) != FLG_IS_INSTRMRG) ||
2784             (sdp->sd_isc->is_osdesc != osp))
2785             continue;

2787         /*
2788          * Remember symbol for use in the third pass. There is no
2789          * reason to save a given symbol more than once, so we take
2790          * advantage of the fact that relocations to a given symbol
2791          * tend to cluster in the list. If this is the same symbol
2792          * we saved last time, don't bother.
2793          */
2794         if (last_sdp != sdp) {
2795             if (aplist_append(sym_alpp, sdp, AL_CNT_STRMRGSYM) ==
2796                 NULL)
2797                 return (0);
2798             last_sdp = sdp;
2799         }

2801         /* Enter the string into our new string table */
2802         name = strmerge_get_reloc_str(of1, rsp);
2803         if (st_insert(mstrtab, name) == -1)
2804             return (0);

2806         /*
2807          * If this is an STT_SECTION symbol, then the second pass
2808          * will need to modify this relocation, so hang on to it.
2809          */
2810         if ((ELF_ST_TYPE(sdp->sd_sym->st_info) == STT_SECTION) &&
2811             (aplist_append(rel_alpp, rsp, AL_CNT_STRMRGREL) == NULL))
2812             return (0);
2813     }

2815     return (1);
2816 }

2818 /*
2819  * If the output section has any SHF MERGE|SHF STRINGS input sections,
2820  * replace them with a single merged/compressed input section.
2821  */
2822 * entry:
2823 *   of1 - Output file descriptor
2824 *   osp - Output section descriptor
2825 *   rel_alpp, sym_alpp, - Address of 2 APlists, to be used
2826 *                       for internal processing. On the initial call to
2827 *                       ld_make_strmerge, these list pointers must be NULL.
2828 *   The caller is encouraged to pass the same lists back for
2829 *   successive calls to this function without freeing
2830 *   them in between calls. This causes a single pair of
2831 *   memory allocations to be reused multiple times.
2832 *
2833 * exit:

```

```

2834 *   If section merging is possible, it is done. If no errors are
2835 *   encountered, True (1) is returned. On error, S_ERROR.
2836 *
2837 *   The contents of rel_alpp and sym_alpp on exit are
2838 *   undefined. The caller can free them, or pass them back to a subsequent
2839 *   call to this routine, but should not examine their contents.
2840 */
2841 static uintptr_t
2842 ld_make_strmerge(Of1_desc *of1, Os_desc *osp, APlist **rel_alpp,
2843             APlist **sym_alpp)
2844 {
2845     Str_tbl      *mstrtab;      /* string table for string merge secs */
2846     Is_desc      *mstrsec;      /* Generated string merge section */
2847     Is_desc      *isp;
2848     Shdr         *mstr_shdr;
2849     Elf_Data     *mstr_data;
2850     Sym_desc     *sdp;
2851     Rel_desc     *rsp;
2852     Aliste       idx;
2853     size_t       data_size;
2854     int          st_setstring_status;
2855     size_t       stoff;

2857     /* If string table compression is disabled, there's nothing to do */
2858     if ((of1->o1_flags1 & FLG_OF1_NCSTTAB) != 0)
2859         return (1);

2861     /*
2862      * Pass over the mergeable input sections, and if they haven't
2863      * all been discarded, create a string table.
2864      */
2865     mstrtab = NULL;
2866     for (APLIST_TRAVERSE(osp->os_mstrisdescs, idx, isp) {
2867         if (isdesc_discarded(isp))
2868             continue;

2870         /*
2871          * Input sections of 0 size are dubiously valid since they do
2872          * not even contain the NUL string. Ignore them.
2873          */
2874         if (isp->is_shdr->sh_size == 0)
2875             continue;

2877         /*
2878          * We have at least one non-discarded section.
2879          * Create a string table descriptor.
2880          */
2881         if ((mstrtab = st_new(FLG_STNEW_COMPRESS)) == NULL)
2882             return (S_ERROR);
2883         break;
2884     }

2886     /* If no string table was created, we have no mergeable sections */
2887     if (mstrtab == NULL)
2888         return (1);

2890     /*
2891      * This routine has to make 3 passes:
2892      *
2893      * 1) Examine all relocations, insert strings from relocations
2894      *    to the mergeable input sections into the string table.
2895      * 2) Modify the relocation values to be correct for the
2896      *    new merged section.
2897      * 3) Modify the symbols used by the relocations to reference
2898      *    the new section.
2899      */

```



```

2900 * These passes cannot be combined:
2901 *   - The string table code works in two passes, and all
2902 *     strings have to be loaded in pass one before the
2903 *     offset of any strings can be determined.
2904 *   - Multiple relocations reference a single symbol, so the
2905 *     symbol cannot be modified until all relocations are
2906 *     fixed.
2907 *
2908 * The number of relocations related to section merging is usually
2909 * a mere fraction of the overall active and output relocation lists,
2910 * and the number of symbols is usually a fraction of the number
2911 * of related relocations. We therefore build APlists for the
2912 * relocations and symbols in the first pass, and then use those
2913 * lists to accelerate the operation of pass 2 and 3.
2914 *
2915 * Reinitialize the lists to a completely empty state.
2916 */
2917 apolist_reset(*rel_alpp);
2918 apolist_reset(*sym_alpp);

2920 /*
2921 * Pass 1:
2922 *
2923 * Every relocation related to this output section (and the input
2924 * sections that make it up) is found in either the active, or the
2925 * output relocation list, depending on whether the relocation is to
2926 * be processed by this invocation of the linker, or inserted into the
2927 * output object.
2928 *
2929 * Build lists of relocations and symbols that will need modification,
2930 * and insert the strings they reference into the mstrtab string table.
2931 */
2932 if (strmerge_pass1(ofl, osp, mstrtab, rel_alpp, sym_alpp,
2933 &ofl->ofl_actrels) == 0)
2934     goto return_s_error;
2935 if (strmerge_pass1(ofl, osp, mstrtab, rel_alpp, sym_alpp,
2936 &ofl->ofl_outrels) == 0)
2937     goto return_s_error;

2939 /*
2940 * Get the size of the new input section. Requesting the
2941 * string table size "cooks" the table, and finalizes its contents.
2942 */
2943 data_size = st_getstrtab_sz(mstrtab);

2945 /* Create a new input section to hold the merged strings */
2946 if (new_section_from_template(ofl, isp, data_size,
2947 &mstrsec, &mstr_shdr, &mstr_data) == S_ERROR)
2948     goto return_s_error;
2949 mstrsec->is_flags |= FLG_IS_GNSTRMRG;

2951 /*
2952 * Allocate a data buffer for the new input section.
2953 * Then, associate the buffer with the string table descriptor.
2954 */
2955 if ((mstr_data->d_buf = libld_malloc(data_size)) == NULL)
2956     goto return_s_error;
2957 if (st_setstrbuf(mstrtab, mstr_data->d_buf, data_size) == -1)
2958     goto return_s_error;

2960 /* Add the new section to the output image */
2961 if (ld_place_section(ofl, mstrsec, NULL, osp->os_idendndx, NULL) ==
2962 (Os_desc *)S_ERROR)
2963     goto return_s_error;

2965 */

```

```

2966 * Pass 2:
2967 *
2968 * Revisit the relocation descriptors with STT_SECTION symbols
2969 * that were saved by the first pass. Update each relocation
2970 * record so that the offset it contains is for the new section
2971 * instead of the original.
2972 */
2973 for (APLIST_TRAVERSE(*rel_alpp, idx, rsp)) {
2974     const char *name;

2976     /* Put the string into the merged string table */
2977     name = strmerge_get_reloc_str(ofl, rsp);
2978     st_setstring_status = st_setstring(mstrtab, name, &stoff);
2979     if (st_setstring_status == -1) {
2980         /*
2981          * A failure to insert at this point means that
2982          * something is corrupt. This isn't a resource issue.
2983          */
2984         assert(st_setstring_status != -1);
2985         goto return_s_error;
2986     }

2988     /*
2989     * Alter the relocation to access the string at the
2990     * new offset in our new string table.
2991     *
2992     * For SHT_RELA platforms, it suffices to simply
2993     * update the rel_raddend field of the relocation.
2994     *
2995     * For SHT_REL platforms, the new "addend" value
2996     * needs to be written at the address being relocated.
2997     * However, we can't alter the input sections which
2998     * are mapped readonly, and the output image has not
2999     * been created yet. So, we defer this operation,
3000     * using the rel_raddend field of the relocation
3001     * which is normally 0 on a REL platform, to pass the
3002     * new "addend" value to ld_perform_outreloc() or
3003     * ld_do_activerelocs(). The FLG_REL_NADDEND flag
3004     * tells them that this is the case.
3005     */
3006     if ((rsp->rel_flags & FLG_REL_RELA) == 0) /* REL */
3007         rsp->rel_flags |= FLG_REL_NADDEND;
3008     rsp->rel_raddend = (Sxword)stoff;

3010     /*
3011     * Generate a symbol name string for STT_SECTION symbols
3012     * that might reference our merged section. This shows up
3013     * in debug output and helps show how the relocation has
3014     * changed from its original input section to our merged one.
3015     */
3016     if (ld_stt_section_sym_name(mstrsec) == NULL)
3017         goto return_s_error;
3018     }

3020 /*
3021 * Pass 3:
3022 *
3023 * Modify the symbols referenced by the relocation descriptors
3024 * so that they reference the new input section containing the
3025 * merged strings instead of the original input sections.
3026 */
3027 for (APLIST_TRAVERSE(*sym_alpp, idx, sdp)) {
3028     /*
3029     * If we've already processed this symbol, don't do it
3030     * twice. strmerge_pass1() uses a heuristic (relocations to
3031     * the same symbol clump together) to avoid inserting a

```

```

3032     * given symbol more than once, but repeat symbols in
3033     * the list can occur.
3034     */
3035     if ((sdp->sd_isc->is_flags & FLG_IS_INSTRMRG) == 0)
3036         continue;

3038     if (ELF_ST_TYPE(sdp->sd_sym->st_info) != STT_SECTION) {
3039         /*
3040          * This is not an STT_SECTION symbol, so its
3041          * value is the offset of the string within the
3042          * input section. Update the address to reflect
3043          * the address in our new merged section.
3044          */
3045         const char *name = sdp->sd_sym->st_value +
3046             (char *)sdp->sd_isc->is_indata->d_buf;

3048         st_setstring_status =
3049             st_setstring(mstrtab, name, &stoff);
3050         if (st_setstring_status == -1) {
3051             /*
3052              * A failure to insert at this point means
3053              * something is corrupt. This isn't a
3054              * resource issue.
3055              */
3056             assert(st_setstring_status != -1);
3057             goto return_s_error;
3058         }

3060         if (ld_sym_copy(sdp) == S_ERROR)
3061             goto return_s_error;
3062         sdp->sd_sym->st_value = (Word)stoff;
3063     }

3065     /* Redirect the symbol to our new merged section */
3066     sdp->sd_isc = mstrsec;
3067 }

3069 /*
3070  * There are no references left to the original input string sections.
3071  * Mark them as discarded so they don't go into the output image.
3072  * At the same time, add up the sizes of the replaced sections.
3073  */
3074 data_size = 0;
3075 for (APLIST_TRAVERSE(osp->os_mstrisdescs, idx, isp)) {
3076     if (isp->is_flags & (FLG_IS_DISCARD | FLG_IS_GNSTRMRG))
3077         continue;

3079     data_size += isp->is_indata->d_size;

3081     isp->is_flags |= FLG_IS_DISCARD;
3082     DBG_CALL(DBG_sec_discarded(ofl->ofl_lml, isp, mstrsec));
3083 }

3085 /* Report how much space we saved in the output section */
3086 DBG_CALL(DBG_sec_genstr_compress(ofl->ofl_lml, osp->os_name, data_size,
3087     mstr_data->d_size));

3089     st_destroy(mstrtab);
3090     return (1);

3092 return_s_error:
3093     st_destroy(mstrtab);
3094     return (S_ERROR);
3095 }

3097 /*

```

```

3098     * Update a data buffers size. A number of sections have to be created, and
3099     * the sections header contributes to the size of the eventual section. Thus,
3100     * a section may be created, and once all associated sections have been created,
3101     * we return to establish the required section size.
3102     */
3103     inline static void
3104     update_data_size(Obj_desc *osp, ulong_t cnt)
3105     {
3106         Is_desc         *isec = ld_os_first_isdesc(osp);
3107         Elf_Data         *data = isec->is_indata;
3108         Shdr             *shdr = osp->os_shdr;
3109         size_t           size = cnt * shdr->sh_entsize;

3111         shdr->sh_size = (Xword)size;
3112         data->d_size = size;
3113     }

3115 /*
3116  * The following sections are built after all input file processing and symbol
3117  * validation has been carried out. The order is important (because the
3118  * addition of a section adds a new symbol there is a chicken and egg problem
3119  * of maintaining the appropriate counts). By maintaining a known order the
3120  * individual routines can compensate for later, known, additions.
3121  */
3122     uintptr_t
3123     ld_make_sections(Obj_desc *ofl)
3124     {
3125         ofl_flag_t       flags = ofl->ofl_flags;
3126         Sg_desc          *sgp;

3128         /*
3129          * Generate any special sections.
3130          */
3131         if (flags & FLG_OF_ADDVERS)
3132             if (make_comment(ofl) == S_ERROR)
3133                 return (S_ERROR);

3135         if (make_interp(ofl) == S_ERROR)
3136             return (S_ERROR);

3138         /*
3139          * Create a capabilities section if required.
3140          */
3141         if (make_cap(ofl, SHT_SUNW_cap, MSG_ORIG(MSG_SCN_SUNWCAP),
3142             ld_targ_t_id.id_cap) == S_ERROR)
3143             return (S_ERROR);

3145         /*
3146          * Create any init/fini array sections.
3147          */
3148         if (make_array(ofl, SHT_INIT_ARRAY, MSG_ORIG(MSG_SCN_INITARRAY),
3149             ofl->ofl_initarray) == S_ERROR)
3150             return (S_ERROR);

3152         if (make_array(ofl, SHT_FINI_ARRAY, MSG_ORIG(MSG_SCN_FINIARRAY),
3153             ofl->ofl_finiarray) == S_ERROR)
3154             return (S_ERROR);

3156         if (make_array(ofl, SHT_PREINIT_ARRAY, MSG_ORIG(MSG_SCN_PREINITARRAY),
3157             ofl->ofl_preiarray) == S_ERROR)
3158             return (S_ERROR);

3160         /*
3161          * Make the .plt section. This occurs after any other relocation
3162          * sections are generated (see reloc_init()) to ensure that the
3163          * associated relocation section is after all the other relocation

```

```

3164     * sections.
3165     */
3166     if ((ofl->ofl_pltcnt) || (ofl->ofl_pltpad))
3167         if (make_plt(ofl) == S_ERROR)
3168             return (S_ERROR);
3170
3171     /*
3172     * Determine whether any sections or files are not referenced. Under
3173     * -Dunused a diagnostic for any unused components is generated, under
3174     * -zignore the component is removed from the final output.
3175     */
3176     if (DBG_ENABLED || (ofl->ofl_flags1 & FLG_OF1_IGNPRC)) {
3177         if (ignore_section_processing(ofl) == S_ERROR)
3178             return (S_ERROR);
3179     }
3180
3181     /*
3182     * If we have detected a situation in which previously placed
3183     * output sections may have been discarded, perform the necessary
3184     * readjustment.
3185     */
3186     if (ofl->ofl_flags & FLG_OF_ADJOSCNT)
3187         adjust_os_count(ofl);
3188
3189     /*
3190     * Do any of the output sections contain input sections that
3191     * are candidates for string table merging? For each such case,
3192     * we create a replacement section, insert it, and discard the
3193     * originals.
3194     * rel_alpp and sym_alpp are used by ld_make_strmerge()
3195     * for its internal processing. We are responsible for the
3196     * initialization and cleanup, and ld_make_strmerge() handles the rest.
3197     * This allows us to reuse a single pair of memory buffers, allocated
3198     * for this processing, for all the output sections.
3199     */
3200     if ((ofl->ofl_flags1 & FLG_OF1_NCSTTAB) == 0) {
3201         int error_seen = 0;
3202         APLIST *rel_alpp = NULL;
3203         APLIST *sym_alpp = NULL;
3204         Aliste idx1;
3206
3207         for (APLIST_TRAVERSE(ofl->ofl_segs, idx1, sgp)) {
3208             Os_desc *osp;
3209             Aliste idx2;
3210
3211             for (APLIST_TRAVERSE(sgp->sg_osdescs, idx2, osp))
3212                 if ((osp->os_mstrisdescs != NULL) &&
3213                     (ld_make_strmerge(ofl, osp,
3214                                     &rel_alpp, &sym_alpp) ==
3215                      S_ERROR)) {
3216                     error_seen = 1;
3217                     break;
3218                 }
3219             }
3220             if (rel_alpp != NULL)
3221                 libld_free(rel_alpp);
3222             if (sym_alpp != NULL)
3223                 libld_free(sym_alpp);
3224             if (error_seen != 0)
3225                 return (S_ERROR);
3226         }
3227     }
3228     /*
3229     * Add any necessary versioning information.
3230     */

```

```

3230     if (!(flags & FLG_OF_NOVERSEC)) {
3231         if ((flags & FLG_OF_VERNEED) &&
3232             (make_verneed(ofl) == S_ERROR))
3233             return (S_ERROR);
3234         if ((flags & FLG_OF_VERDEF) &&
3235             (make_verdef(ofl) == S_ERROR))
3236             return (S_ERROR);
3237         if ((flags & (FLG_OF_VERNEED | FLG_OF_VERDEF)) &&
3238             ((ofl->ofl_osversym = make_sym_sec(ofl,
3239                 MSG_ORIG(MSG_SCN_SUNWVERSYM), SHT_SUNW_versym,
3240                 ld_targ.t_id.id_version) == (Os_desc*)S_ERROR))
3241             return (S_ERROR);
3242     }
3244     /*
3245     * Create a syminfo section if necessary.
3246     */
3247     if (flags & FLG_OF_SYMINFO) {
3248         if ((ofl->ofl_ossyminfo = make_sym_sec(ofl,
3249             MSG_ORIG(MSG_SCN_SUNWSYMINFO), SHT_SUNW_syminfo,
3250             ld_targ.t_id.id_syminfo) == (Os_desc *)S_ERROR)
3251             return (S_ERROR);
3252     }
3254     if (flags & FLG_OF_COMREL) {
3255         /*
3256         * If -zcombreloc is enabled then all relocations (except for
3257         * the PLT's) are coalesced into a single relocation section.
3258         */
3259         if (ofl->ofl_reloccnt) {
3260             if (make_reloc(ofl, NULL) == S_ERROR)
3261                 return (S_ERROR);
3262         }
3263     } else {
3264         Aliste idx1;
3266
3267         /*
3268         * Create the required output relocation sections. Note, new
3269         * sections may be added to the section list that is being
3270         * traversed. These insertions can move the elements of the
3271         * Alist such that a section descriptor is re-read. Recursion
3272         * is prevented by maintaining a previous section pointer and
3273         * insuring that this pointer isn't re-examined.
3274         */
3275         for (APLIST_TRAVERSE(ofl->ofl_segs, idx1, sgp)) {
3276             Os_desc *osp, *posp = 0;
3277             Aliste idx2;
3278
3279             for (APLIST_TRAVERSE(sgp->sg_osdescs, idx2, osp)) {
3280                 if ((osp != posp) && osp->os_szoutrels &&
3281                     (osp != ofl->ofl_osplt)) {
3282                     if (make_reloc(ofl, osp) == S_ERROR)
3283                         return (S_ERROR);
3284                 }
3285                 posp = osp;
3286             }
3288         }
3289     }
3290     /*
3291     * If we're not building a combined relocation section, then
3292     * build a .rel[a] section as required.
3293     */
3294     if (ofl->ofl_relocrelsz) {
3295         if (make_reloc(ofl, NULL) == S_ERROR)
3296             return (S_ERROR);
3297     }

```

```

3296     }
3298     /*
3299     * The PLT relocations are always in their own section, and we try to
3300     * keep them at the end of the PLT table. We do this to keep the hot
3301     * "data" PLT's at the head of the table nearer the .dynsym & .hash.
3302     */
3303     if (ofl->ofl_osplt && ofl->ofl_relocpltsz) {
3304         if (make_reloc(ofl, ofl->ofl_osplt) == S_ERROR)
3305             return (S_ERROR);
3306     }
3308     /*
3309     * Finally build the symbol and section header sections.
3310     */
3311     if (flags & FLG_OF_DYNAMIC) {
3312         if (make_dynamic(ofl) == S_ERROR)
3313             return (S_ERROR);
3315         /*
3316         * A number of sections aren't necessary within a relocatable
3317         * object, even if -dy has been used.
3318         */
3319         if (!(flags & FLG_OF_RELOBJ)) {
3320             if (make_hash(ofl) == S_ERROR)
3321                 return (S_ERROR);
3322             if (make_dynstr(ofl) == S_ERROR)
3323                 return (S_ERROR);
3324             if (make_dynsym(ofl) == S_ERROR)
3325                 return (S_ERROR);
3326             if (ld_unwind_make_hdr(ofl) == S_ERROR)
3327                 return (S_ERROR);
3328             if (make_dynsort(ofl) == S_ERROR)
3329                 return (S_ERROR);
3330         }
3331     }
3333     if (!(flags & FLG_OF_STRIP) || (flags & FLG_OF_RELOBJ) ||
3334         ((flags & FLG_OF_STATIC) && ofl->ofl_osversym)) {
3335         /*
3336         * Do we need to make a SHT_SYMTAB_SHNDX section
3337         * for the dynsym. If so - do it now.
3338         */
3339         if (ofl->ofl_osdynsym &&
3340             ((ofl->ofl_shdrcnt + 3) >= SHN_LORESERVE)) {
3341             if (make_dynsym_shndx(ofl) == S_ERROR)
3342                 return (S_ERROR);
3343         }
3345         if (make_strtab(ofl) == S_ERROR)
3346             return (S_ERROR);
3347         if (make_symtab(ofl) == S_ERROR)
3348             return (S_ERROR);
3349     } else {
3350         /*
3351         * Do we need to make a SHT_SYMTAB_SHNDX section
3352         * for the dynsym. If so - do it now.
3353         */
3354         if (ofl->ofl_osdynsym &&
3355             ((ofl->ofl_shdrcnt + 1) >= SHN_LORESERVE)) {
3356             if (make_dynsym_shndx(ofl) == S_ERROR)
3357                 return (S_ERROR);
3358         }
3359     }
3361     if (make_shstrtab(ofl) == S_ERROR)

```

```

3362         return (S_ERROR);
3364     /*
3365     * Now that we've created all output sections, adjust the size of the
3366     * SHT_SUNW_versym and SHT_SUNW_syminfo section, which are dependent on
3367     * the associated symbol table sizes.
3368     */
3369     if (ofl->ofl_osversym || ofl->ofl_ossyminfo) {
3370         ulong_t cnt;
3371         Is_desc *isp;
3372         Os_desc *osp;
3374         if (OFL_IS_STATIC_OBJ(ofl))
3375             osp = ofl->ofl_ossymtab;
3376         else
3377             osp = ofl->ofl_osdynsym;
3379         isp = ld_os_first_isdesc(osp);
3380         cnt = (isp->is_shdr->sh_size / isp->is_shdr->sh_entsize);
3382         if (ofl->ofl_osversym)
3383             update_data_size(ofl->ofl_osversym, cnt);
3385         if (ofl->ofl_ossyminfo)
3386             update_data_size(ofl->ofl_ossyminfo, cnt);
3387     }
3389     /*
3390     * Now that we've created all output sections, adjust the size of the
3391     * SHT_SUNW_capinfo, which is dependent on the associated symbol table
3392     * size.
3393     */
3394     if (ofl->ofl_oscapiinfo) {
3395         ulong_t cnt;
3397         /*
3398         * Symbol capabilities symbols are placed directly after the
3399         * STT_FILE symbol, section symbols, and any register symbols.
3400         * Effectively these are the first of any series of demoted
3401         * (scoped) symbols.
3402         */
3403         if (OFL_IS_STATIC_OBJ(ofl))
3404             cnt = SYMTAB_ALL_CNT(ofl);
3405         else
3406             cnt = DYNYSYM_ALL_CNT(ofl);
3408         update_data_size(ofl->ofl_oscapiinfo, cnt);
3409     }
3410     return (1);
3411 }
3413 /*
3414 * Build an additional data section - used to back OBJT symbol definitions
3415 * added with a mapfile.
3416 */
3417 Is_desc *
3418 ld_make_data(Ofld_desc *ofl, size_t size)
3419 {
3420     Shdr *shdr;
3421     Elf_Data *data;
3422     Is_desc *isec;
3424     if (new_section(ofl, SHT_PROGBITS, MSG_ORIG(MSG_SCN_DATA), 0,
3425         &isec, &shdr, &data) == S_ERROR)
3426         return ((Is_desc *)S_ERROR);

```

```

3428     data->d_size = size;
3429     shdr->sh_size = (Xword)size;
3430     shdr->sh_flags |= SHF_WRITE;

3432     if (aplist_append(&ofl->ofl_mapdata, isec, AL_CNT_OFL_MAPSECS) == NULL)
3433         return ((Is_desc *)S_ERROR);

3435     return (isec);
3436 }

3438 /*
3439  * Build an additional text section - used to back FUNC symbol definitions
3440  * added with a mapfile.
3441  */
3442 Is_desc *
3443 ld_make_text(Ofld_desc *ofl, size_t size)
3444 {
3445     Shdr      *shdr;
3446     Elf_Data  *data;
3447     Is_desc   *isec;

3449     /*
3450      * Insure the size is sufficient to contain the minimum return
3451      * instruction.
3452      */
3453     if (size < ld_targ.t_nf.nf_size)
3454         size = ld_targ.t_nf.nf_size;

3456     if (new_section(ofl, SHT_PROGBITS, MSG_ORIG(MSG_SCN_TEXT), 0,
3457                    &isec, &shdr, &data) == S_ERROR)
3458         return ((Is_desc *)S_ERROR);

3460     data->d_size = size;
3461     shdr->sh_size = (Xword)size;
3462     shdr->sh_flags |= SHF_EXECINSTR;

3464     /*
3465      * Fill the buffer with the appropriate return instruction.
3466      * Note that there is no need to swap bytes on a non-native,
3467      * link, as the data being copied is given in bytes.
3468      */
3469     if ((data->d_buf = libld_calloc(size, 1)) == NULL)
3470         return ((Is_desc *)S_ERROR);
3471     (void) memcpy(data->d_buf, ld_targ.t_nf.nf_template,
3472                 ld_targ.t_nf.nf_size);

3474     /*
3475      * If size was larger than required, and the target supplies
3476      * a fill function, use it to fill the balance. If there is no
3477      * fill function, we accept the 0-fill supplied by libld_calloc().
3478      */
3479     if ((ld_targ.t_ff.ff_execfill != NULL) && (size > ld_targ.t_nf.nf_size))
3480         ld_targ.t_ff.ff_execfill(data->d_buf, ld_targ.t_nf.nf_size,
3481                                 size - ld_targ.t_nf.nf_size);

3483     if (aplist_append(&ofl->ofl_maptxt, isec, AL_CNT_OFL_MAPSECS) == NULL)
3484         return ((Is_desc *)S_ERROR);

3486     return (isec);
3487 }

3489 void
3490 ld_comdat_validate(Ofld_desc *ofl, Ifld_desc *ifld)
3491 {
3492     int i;

```

```

3494     for (i = 0; i < ifld->ifld_shnum; i++) {
3495         Is_desc *isp = ifld->ifld_idesc[i];
3496         int types = 0;
3497         char buf[1024] = "";
3498         Group_desc *gr = NULL;

3500         if ((isp == NULL) || (isp->is_flags & FLG_IS_COMDAT) == 0)
3501             continue;

3503         if (isp->is_shdr->sh_type == SHT_SUNW_COMDAT) {
3504             types++;
3505             (void) strcpy(buf, MSG_ORIG(MSG_STR_SUNW_COMDAT),
3506                          sizeof (buf));
3507         }

3509         if (strcmp(MSG_ORIG(MSG_SCN_GNU_LINKONCE), isp->is_name,
3510                  MSG_SCN_GNU_LINKONCE_SIZE) == 0) {
3511             types++;
3512             if (types > 1)
3513                 (void) strcat(buf, ", ", sizeof (buf));
3514             (void) strcat(buf, MSG_ORIG(MSG_SCN_GNU_LINKONCE),
3515                          sizeof (buf));
3516         }

3518         if ((isp->is_shdr->sh_flags & SHF_GROUP) &&
3519             ((gr = ld_get_group(ofl, isp)) != NULL) &&
3520             (gr->gd_data[0] & GRP_COMDAT)) {
3521             types++;
3522             if (types > 1)
3523                 (void) strcat(buf, ", ", sizeof (buf));
3524             (void) strcat(buf, MSG_ORIG(MSG_STR_GROUP),
3525                          sizeof (buf));
3526         }

3528         if (types > 1)
3529             ld_eprintf(ofl, ERR_FATAL,
3530                       MSG_INTL(MSG_SCN_MULTICOMDAT), ifld->ifld_name,
3531                       EC_WORD(isp->is_scndx), isp->is_name, buf);
3532     }
3533 }

```

```

*****
97621 Fri Mar 1 17:10:03 2019
new/usr/src/cmd/sgs/libld/common/syms.c
code review from Robert
*****
_____unchanged_portion_omitted_____

954 /*
955  * At this point all symbol input processing has been completed, therefore
956  * complete the symbol table entries by generating any necessary internal
957  * symbols.
958  */
959 uintptr_t
960 ld_sym_spec(Of1_desc *of1)
961 {
962     Sym_desc      *sdp;
963     Sg_desc       *sgp;
964     Aliste        idx1;

965     DBG_CALL(DBG_syms_spec_title(of1->of1_lml));

966     /*
967     * For each section in the output file, look for symbols named for the
968     * __start/__stop patterns.  If references exist, flesh the symbols to
969     * be defined.
970     *
971     * The symbols are given values at the same time as the other special
972     * symbols.
973     */
974     if (!(of1->of1_flags & FLG_OF_RELOBJ) ||
975         (of1->of1_flags & FLG_OF_KMOD)) {
976         Aliste        idx1;

979 #endif /* ! codereview */
980         for (APLIST_TRAVERSE(of1->of1_segs, idx1, sgp)) {
981             Os_desc *osp;
982             Aliste idx2;

984             for (APLIST_TRAVERSE(sgp->sg_osdescs, idx2, osp)) {
985                 if (is_cname(osp->os_name)) {
986                     sym_add_bounds(of1, osp,
987                                   SDAUX_ID_SECBOUND_START);
988                     sym_add_bounds(of1, osp,
989                                   SDAUX_ID_SECBOUND_STOP);
990                 }
991             }
992         }
993     }

995     if (of1->of1_flags & FLG_OF_RELOBJ)
996         return (1);

998     if (sym_add_spec(MSG_ORIG(MSG_SYM_ETEXT), MSG_ORIG(MSG_SYM_ETEXT_U),
999                   SDAUX_ID_ETEXT, 0, (FLG_SY_DEFAULT | FLG_SY_EXPDEF),
1000                   of1) == S_ERROR)
1001         return (S_ERROR);
1002     if (sym_add_spec(MSG_ORIG(MSG_SYM_EDATA), MSG_ORIG(MSG_SYM_EDATA_U),
1003                   SDAUX_ID_EDATA, 0, (FLG_SY_DEFAULT | FLG_SY_EXPDEF),
1004                   of1) == S_ERROR)
1005         return (S_ERROR);
1006     if (sym_add_spec(MSG_ORIG(MSG_SYM_END), MSG_ORIG(MSG_SYM_END_U),
1007                   SDAUX_ID_END, FLG_SY_DYNSORT, (FLG_SY_DEFAULT | FLG_SY_EXPDEF),
1008                   of1) == S_ERROR)
1009         return (S_ERROR);
1010     if (sym_add_spec(MSG_ORIG(MSG_SYM_L_END), MSG_ORIG(MSG_SYM_L_END_U),
1011                   SDAUX_ID_END, 0, FLG_SY_HIDDEN, of1) == S_ERROR)

```

```

1012         return (S_ERROR);
1013     if (sym_add_spec(MSG_ORIG(MSG_SYM_L_START), MSG_ORIG(MSG_SYM_L_START_U),
1014                   SDAUX_ID_START, 0, FLG_SY_HIDDEN, of1) == S_ERROR)
1015         return (S_ERROR);

1017     /*
1018     * Historically we've always produced a _DYNAMIC symbol, even for
1019     * static executables (in which case its value will be 0).
1020     */
1021     if (sym_add_spec(MSG_ORIG(MSG_SYM_DYNAMIC), MSG_ORIG(MSG_SYM_DYNAMIC_U),
1022                   SDAUX_ID_DYN, FLG_SY_DYNSORT, (FLG_SY_DEFAULT | FLG_SY_EXPDEF),
1023                   of1) == S_ERROR)
1024         return (S_ERROR);

1026     if (OFL_ALLOW_DYNSYM(of1))
1027         if (sym_add_spec(MSG_ORIG(MSG_SYM_PLKTBL),
1028                       MSG_ORIG(MSG_SYM_PLKTBL_U), SDAUX_ID_PLT,
1029                       FLG_SY_DYNSORT, (FLG_SY_DEFAULT | FLG_SY_EXPDEF),
1030                       of1) == S_ERROR)
1031             return (S_ERROR);

1033     /*
1034     * A GOT reference will be accompanied by the associated GOT symbol.
1035     * Make sure it gets assigned the appropriate special attributes.
1036     */
1037     if (((sdp = ld_sym_find(MSG_ORIG(MSG_SYM_GOTFBL_U),
1038                           SYM_NOHASH, NULL, of1)) != NULL) && (sdp->sd_ref != REF_DYN_SEEN)) {
1039         if (sym_add_spec(MSG_ORIG(MSG_SYM_GOTFBL),
1040                       MSG_ORIG(MSG_SYM_GOTFBL_U), SDAUX_ID_GOT, FLG_SY_DYNSORT,
1041                       (FLG_SY_DEFAULT | FLG_SY_EXPDEF), of1) == S_ERROR)
1042             return (S_ERROR);
1043     }

1045     return (1);
1046 }

1048 /*
1049  * Determine a potential capability symbol's visibility.
1050  *
1051  * The -z symbolcap option transforms an object capabilities relocatable object
1052  * into a symbol capabilities relocatable object. Any global function symbols,
1053  * or initialized global data symbols are candidates for transforming into local
1054  * symbol capabilities definitions. However, if a user indicates that a symbol
1055  * should be demoted to local using a mapfile, then there is no need to
1056  * transform the associated global symbol.
1057  *
1058  * Normally, a symbol's visibility is determined after the symbol resolution
1059  * process, after all symbol state has been gathered and resolved. However,
1060  * for -z symbolcap, this determination is too late. When a global symbol is
1061  * read from an input file we need to determine it's visibility so as to decide
1062  * whether to create a local or not.
1063  *
1064  * If a user has explicitly defined this symbol as having local scope within a
1065  * mapfile, then a symbol of the same name already exists. However, explicit
1066  * local definitions are uncommon, as most mapfiles define the global symbol
1067  * requirements together with an auto-reduction directive '*'. If this state
1068  * has been defined, then we must make sure that the new symbol isn't a type
1069  * that can not be demoted to local.
1070  */
1071 static int
1072 sym_cap_vis(const char *name, Word hash, Sym *sym, Of1_desc *of1)
1073 {
1074     Sym_desc      *sdp;
1075     uchar_t       vis;
1076     avl_index_t   where;
1077     sd_flag_t     sdflags = 0;

```

```

1079  /*
1080  * Determine the visibility of the new symbol.
1081  */
1082  vis = ELF_ST_VISIBILITY(sym->st_other);
1083  switch (vis) {
1084  case STV_EXPORTED:
1085      sdflags |= FLG_SY_EXPORT;
1086      break;
1087  case STV_SINGLETON:
1088      sdflags |= FLG_SY_SINGLE;
1089      break;
1090  }

1092  /*
1093  * Determine whether a symbol definition already exists, and if so
1094  * obtain the visibility.
1095  */
1096  if ((sdp = ld_sym_find(name, hash, &where, ofl)) != NULL)
1097      sdflags |= sdp->sd_flags;

1099  /*
1100  * Determine whether the symbol flags indicate this symbol should be
1101  * hidden.
1102  */
1103  if ((ofl->ofl_flags & (FLG_OF_AUTOLCL | FLG_OF_AUTOELM)) &&
1104      ((sdflags & MSK_SY_NOAUTO) == 0))
1105      sdflags |= FLG_SY_HIDDEN;

1107  return ((sdflags & FLG_SY_HIDDEN) == 0);
1108 }

1110 /*
1111 * This routine checks to see if a symbols visibility needs to be reduced to
1112 * either SYMBOLIC or LOCAL. This routine can be called from either
1113 * reloc_init() or sym_validate().
1114 */
1115 void
1116 ld_sym_adjust_vis(Sym_desc *sdp, Of1_desc *ofl)
1117 {
1118     ofl_flag_t    oflags = ofl->ofl_flags;
1119     Sym           *sym = sdp->sd_sym;

1121     if ((sdp->sd_ref == REF_REL_NEED) &&
1122         (sdp->sd_sym->st_shndx != SHN_UNDEF)) {
1123         /*
1124          * If auto-reduction/elimination is enabled, reduce any
1125          * non-versioned, and non-local capabilities global symbols.
1126          * A symbol is a candidate for auto-reduction/elimination if:
1127          *
1128          * - the symbol wasn't explicitly defined within a mapfile
1129          *   (in which case all the necessary state has been applied
1130          *   to the symbol), or
1131          * - the symbol isn't one of the family of reserved
1132          *   special symbols (ie. _end, _etext, etc.), or
1133          * - the symbol isn't a SINGLETON, or
1134          * - the symbol wasn't explicitly defined within a version
1135          *   definition associated with an input relocatable object.
1136          *
1137          * Indicate that the symbol has been reduced as it may be
1138          * necessary to print these symbols later.
1139          */
1140         if ((oflags & (FLG_OF_AUTOLCL | FLG_OF_AUTOELM)) &&
1141             ((sdp->sd_flags & MSK_SY_NOAUTO) == 0)) {
1142             if ((sdp->sd_flags & FLG_SY_HIDDEN) == 0) {
1143                 sdp->sd_flags |=

```

```

1144         (FLG_SY_REduced | FLG_SY_HIDDEN);
1145     }

1147     if (oflags & (FLG_OF_REDLSYM | FLG_OF_AUTOELM)) {
1148         sdp->sd_flags |= FLG_SY_ELIM;
1149         sym->st_other = STV_ELIMINATE |
1150             (sym->st_other & ~MSK_SYM_VISIBILITY);
1151     } else if (ELF_ST_VISIBILITY(sym->st_other) !=
1152               STV_INTERNAL)
1153         sym->st_other = STV_HIDDEN |
1154             (sym->st_other & ~MSK_SYM_VISIBILITY);
1155     }

1157     /*
1158     * If -Bsymbolic is in effect, and the symbol hasn't explicitly
1159     * been defined nodirect (via a mapfile), then bind the global
1160     * symbol symbolically and assign the STV_PROTECTED visibility
1161     * attribute.
1162     */
1163     if ((oflags & FLG_OF_SYMBOLIC) &&
1164         ((sdp->sd_flags & (FLG_SY_HIDDEN | FLG_SY_NDIR)) == 0)) {
1165         sdp->sd_flags |= FLG_SY_PROTECT;
1166         if (ELF_ST_VISIBILITY(sym->st_other) == STV_DEFAULT)
1167             sym->st_other = STV_PROTECTED |
1168                 (sym->st_other & ~MSK_SYM_VISIBILITY);
1169     }
1170 }

1172     /*
1173     * Indicate that this symbol has had it's visibility checked so that
1174     * we don't need to do this investigation again.
1175     */
1176     sdp->sd_flags |= FLG_SY_VISIBLE;
1177 }

1179 /*
1180 * Make sure a symbol definition is local to the object being built.
1181 */
1182 inline static int
1183 ensure_sym_local(Of1_desc *ofl, Sym_desc *sdp, const char *str)
1184 {
1185     if (sdp->sd_sym->st_shndx == SHN_UNDEF) {
1186         if (str) {
1187             ld_eprintf(ofl, ERR_FATAL, MSG_INTL(MSG_SYM_UNDEF),
1188                       str, demangle((char *)sdp->sd_name));
1189         }
1190         return (1);
1191     }
1192     if (sdp->sd_ref != REF_REL_NEED) {
1193         if (str) {
1194             ld_eprintf(ofl, ERR_FATAL, MSG_INTL(MSG_SYM_EXTERN),
1195                       str, demangle((char *)sdp->sd_name),
1196                       sdp->sd_file->ifl_name);
1197         }
1198         return (1);
1199     }

1201     sdp->sd_flags |= FLG_SY_UPREQD;
1202     if (sdp->sd_isc) {
1203         sdp->sd_isc->is_flags |= FLG_IS_SECTREF;
1204         sdp->sd_isc->is_file->ifl_flags |= FLG_IF_FILEREF;
1205     }
1206     return (0);
1207 }

1209 /*

```

```

1210 * Make sure all the symbol definitions required for inittarray, finiarray, or
1211 * preinitarray's are local to the object being built.
1212 */
1213 static int
1214 ensure_array_local(Of1_desc *of1, APlist *apl, const char *str)
1215 {
1216     Aliste      idx;
1217     Sym_desc    *sdp;
1218     int         ret = 0;
1219
1220     for (APLIST_TRAVERSE(apl, idx, sdp))
1221         ret += ensure_sym_local(of1, sdp, str);
1222
1223     return (ret);
1224 }
1225
1226 /*
1227 * After all symbol table input processing has been finished, and all relocation
1228 * counting has been carried out (ie. no more symbols will be read, generated,
1229 * or modified), validate and count the relevant entries:
1230 *
1231 * - check and print any undefined symbols remaining. Note that if a symbol
1232 *   has been defined by virtue of the inclusion of an implicit shared
1233 *   library, it is still classed as undefined.
1234 *
1235 * - count the number of global needed symbols together with the size of
1236 *   their associated name strings (if scoping has been indicated these
1237 *   symbols may be reduced to locals).
1238 *
1239 * - establish the size and alignment requirements for the global .bss
1240 *   section (the alignment of this section is based on the first symbol
1241 *   that it will contain).
1242 */
1243 uintptr_t
1244 ld_sym_validate(Of1_desc *of1)
1245 {
1246     Sym_avlnode *sav;
1247     Sym_desc    *sdp;
1248     Sym         *sym;
1249     ofl_flag_t  ofl_flags = of1->ofl_flags;
1250     ofl_flag_t  undef = 0, needed = 0, verdesc = 0;
1251     Xword       bssalign = 0, tlbssalign = 0;
1252     Boolean     need_bss, need_tlbss;
1253     Xword       bsssize = 0, tlbsssize = 0;
1254     #if defined(_ELF64)
1255     Xword       lbssalign = 0, lbsssize = 0;
1256     Boolean     need_lbss;
1257     #endif
1258     int         ret, allow_ldynsym;
1259     uchar_t     type;
1260     ofl_flag_t  undef_state = 0;
1261
1262     DBG_CALL(DBG_basic_validate(of1->ofl_lml));
1263
1264     /*
1265     * The need_XXX booleans are used to determine whether we need to
1266     * create each type of bss section. We used to create these sections
1267     * if the sum of the required sizes for each type were non-zero.
1268     * However, it is possible for a compiler to generate COMMON variables
1269     * of zero-length and this tricks that logic --- even zero-length
1270     * symbols need an output section.
1271     */
1272     need_bss = need_tlbss = FALSE;
1273     #if defined(_ELF64)
1274     need_lbss = FALSE;
1275     #endif

```

```

1277     /*
1278     * Determine how undefined symbols are handled:
1279     *
1280     * fatal:
1281     *   If this link-edit calls for no undefined symbols to remain
1282     *   (this is the default case when generating an executable but
1283     *   can be enforced for any object using -z defs), a fatal error
1284     *   condition will be indicated.
1285     *
1286     * warning:
1287     *   If we're creating a shared object, and either the -Bsymbolic
1288     *   flag is set, or the user has turned on the -z guidance feature,
1289     *   then a non-fatal warning is issued for each symbol.
1290     *
1291     * ignore:
1292     *   In all other cases, undefined symbols are quietly allowed.
1293     */
1294     if (oflags & FLG_OF_NOUNDEF) {
1295         undef = FLG_OF_FATAL;
1296     } else if (oflags & FLG_OF_SHAROBJ) {
1297         if ((oflags & FLG_OF_SYMBOLIC) ||
1298             OFL_GUIDANCE(of1, FLG_OFG_NO_DEFS))
1299             undef = FLG_OF_WARN;
1300     }
1301
1302     /*
1303     * If the symbol is referenced from an implicitly included shared object
1304     * (ie. it's not on the NEEDED list) then the symbol is also classified
1305     * as undefined and a fatal error condition will be indicated.
1306     */
1307     if ((oflags & FLG_OF_NOUNDEF) || !(oflags & FLG_OF_SHAROBJ))
1308         needed = FLG_OF_FATAL;
1309     else if ((oflags & FLG_OF_SHAROBJ) &&
1310             OFL_GUIDANCE(of1, FLG_OFG_NO_DEFS))
1311         needed = FLG_OF_WARN;
1312
1313     /*
1314     * If the output image is being versioned, then all symbol definitions
1315     * must be associated with a version. Any symbol that isn't associated
1316     * with a version is classified as undefined, and a fatal error
1317     * condition is indicated.
1318     */
1319     if ((oflags & FLG_OF_VERDEF) && (of1->ofl_vercnt > VER_NDX_GLOBAL))
1320         verdesc = FLG_OF_FATAL;
1321
1322     allow_ldynsym = OFL_ALLOW_LDYNSYM(of1);
1323
1324     if (allow_ldynsym) {
1325         /*
1326         * Normally, we disallow symbols with 0 size from appearing
1327         * in a dyn[sym|tls]sort section. However, there are some
1328         * symbols that serve special purposes that we want to exempt
1329         * from this rule. Look them up, and set their
1330         * FLG_SY_DYNSORT flag.
1331         */
1332         static const char *special[] = {
1333             MSG_ORIG(MSG_SYM_INIT_U),      /* _init */
1334             MSG_ORIG(MSG_SYM_FINI_U),      /* _fini */
1335             MSG_ORIG(MSG_SYM_START),       /* _start */
1336             NULL
1337         };
1338         int i;
1339
1340         for (i = 0; special[i] != NULL; i++) {
1341             if ((sdp = ld_sym_find(special[i],

```



```

1342         SYM_NOHASH, NULL, ofl)) != NULL) &&
1343         (sdp->sd_sym->st_size == 0)) {
1344             if (ld_sym_copy(sdp) == S_ERROR)
1345                 return (S_ERROR);
1346             sdp->sd_flags |= FLG_SY_DYNSORT;
1347         }
1348     }
1349 }
1350
1351 /*
1352  * Collect and validate the globals from the internal symbol table.
1353  */
1354 for (sav = avl_first(&ofl->ofl_symavl); sav;
1355      sav = AVL_NEXT(&ofl->ofl_symavl, sav)) {
1356     Is_desc      *isp;
1357     int          undeferr = 0;
1358     uchar_t      vis;
1359
1360     sdp = sav->sav_sdp;
1361
1362     /*
1363      * If undefined symbols are allowed, and we're not being
1364      * asked to supply guidance, ignore any symbols that are
1365      * not needed.
1366      */
1367     if (!(oflags & FLG_OF_NOUNDEF) &&
1368         !OFL_GUIDANCE(ofl, FLG_OFG_NO_DEFS) &&
1369         (sdp->sd_ref == REF_DYN_SEEN))
1370         continue;
1371
1372     /*
1373      * If the symbol originates from an external or parent mapfile
1374      * reference and hasn't been matched to a reference from a
1375      * relocatable object, ignore it.
1376      */
1377     if ((sdp->sd_flags & (FLG_SY_EXTERN | FLG_SY_PARENT)) &&
1378         ((sdp->sd_flags & FLG_SY_MAPUSED) == 0)) {
1379         sdp->sd_flags |= FLG_SY_INVALID;
1380         continue;
1381     }
1382
1383     sym = sdp->sd_sym;
1384     type = ELF_ST_TYPE(sym->st_info);
1385
1386     /*
1387      * Sanity check TLS.
1388      */
1389     if ((type == STT_TLS) && (sym->st_size != 0) &&
1390         (sym->st_shndx != SHN_UNDEF) &&
1391         (sym->st_shndx != SHN_COMMON)) {
1392         Is_desc      *isp = sdp->sd_isc;
1393         Ifl_desc     *ifl = sdp->sd_file;
1394
1395         if ((isp == NULL) || (isp->is_shdr == NULL) ||
1396             ((isp->is_shdr->sh_flags & SHF_TLS) == 0)) {
1397             ld_eprintf(ofl, ERR_FATAL,
1398                 MSG_INTL(MSG_SYM_TLS),
1399                 demangle(sdp->sd_name), ifl->ifl_name);
1400             continue;
1401         }
1402     }
1403
1404     if ((sdp->sd_flags & FLG_SY_VISIBLE) == 0)
1405         ld_sym_adjust_vis(sdp, ofl);
1406
1407     if ((sdp->sd_flags & FLG_SY_REduced) &&

```

```

1408         (oflags & FLG_OF_PROCDRED)) {
1409             DBG_CALL(DBG_syms_reduce(ofl, DBG_SYM_REDUCE_GLOBAL,
1410                 sdp, 0, 0));
1411         }
1412
1413     /*
1414      * Record any STV_SINGLETON existence.
1415      */
1416     if ((vis = ELF_ST_VISIBILITY(sym->st_other)) == STV_SINGLETON)
1417         ofl->ofl_dtflags_1 |= DF_1_SINGLETON;
1418
1419     /*
1420      * If building a shared object or executable, and this is a
1421      * non-weak UNDEF symbol with reduced visibility (STV_*), then
1422      * give a fatal error.
1423      */
1424     if (((oflags & FLG_OF_RELOBJ) == 0) &&
1425         (sym->st_shndx == SHN_UNDEF) &&
1426         (ELF_ST_BIND(sym->st_info) != STB_WEAK)) {
1427         if (vis && (vis != STV_SINGLETON)) {
1428             sym_undef_entry(ofl, sdp, BNDLOCAL,
1429                 FLG_OF_FATAL, &undef_state);
1430             continue;
1431         }
1432     }
1433
1434     /*
1435      * If this symbol is defined in a non-allocatable section,
1436      * reduce it to local symbol.
1437      */
1438     if (((isp = sdp->sd_isc) != 0) && isp->is_shdr &&
1439         ((isp->is_shdr->sh_flags & SHF_ALLOC) == 0)) {
1440         sdp->sd_flags |= (FLG_SY_REduced | FLG_SY_HIDDEN);
1441     }
1442
1443     /*
1444      * If this symbol originated as a SHN_SUNW_IGNORE, it will have
1445      * been processed as an SHN_UNDEF. Return the symbol to its
1446      * original index for validation, and propagation to the output
1447      * file.
1448      */
1449     if (sdp->sd_flags & FLG_SY_IGNORE)
1450         sdp->sd_shndx = SHN_SUNW_IGNORE;
1451
1452     if (undef) {
1453         /*
1454          * If a non-weak reference remains undefined, or if a
1455          * mapfile reference is not bound to the relocatable
1456          * objects that make up the object being built, we have
1457          * a fatal error.
1458          *
1459          * The exceptions are symbols which are defined to be
1460          * found in the parent (FLG_SY_PARENT), which is really
1461          * only meaningful for direct binding, or are defined
1462          * external (FLG_SY_EXTERN) so as to suppress -zdefs
1463          * errors.
1464          *
1465          * Register symbols are always allowed to be UNDEF.
1466          *
1467          * Note that we don't include references created via -u
1468          * in the same shared object binding test. This is for
1469          * backward compatibility, in that a number of archive
1470          * makefile rules used -u to cause archive extraction.
1471          * These same rules have been cut and pasted to apply
1472          * to shared objects, and thus although the -u reference
1473          * is redundant, flagging it as fatal could cause some

```

```

1474     * build to fail. Also we have documented the use of
1475     * -u as a mechanism to cause binding to weak version
1476     * definitions, thus giving users an error condition
1477     * would be incorrect.
1478     */
1479     if (!(sdp->sd_flags & FLG_SY_REGSYM) &&
1480         ((sym->st_shndx == SHN_UNDEF) &&
1481          ((ELF_ST_BIND(sym->st_info) != STB_WEAK) &&
1482           ((sdp->sd_flags &
1483            (FLG_SY_PARENT | FLG_SY_EXTERN)) == 0)) ||
1484           ((sdp->sd_flags &
1485            (FLG_SY_MAPREF | FLG_SY_MAPUSED | FLG_SY_HIDDEN |
1486             FLG_SY_PROTECT)) == FLG_SY_MAPREF))) {
1487         sym_undef_entry(of1, sdp, UNDEF, undef,
1488                       &undef_state);
1489         undeferr = 1;
1490     }
1491 } else {
1492     /*
1493     * For building things like shared objects (or anything
1494     * -znodefs), undefined symbols are allowed.
1495     *
1496     * If a mapfile reference remains undefined the user
1497     * would probably like a warning at least (they've
1498     * usually mis-spelt the reference). Refer to the above
1499     * comments for discussion on -u references, which
1500     * are not tested for in the same manner.
1501     */
1502     if ((sdp->sd_flags &
1503         (FLG_SY_MAPREF | FLG_SY_MAPUSED)) ==
1504         FLG_SY_MAPREF) {
1505         sym_undef_entry(of1, sdp, UNDEF, FLG_OF_WARN,
1506                       &undef_state);
1507         undeferr = 1;
1508     }
1509 }
1510
1511 /*
1512 * If this symbol comes from a dependency mark the dependency
1513 * as required (-z ignore can result in unused dependencies
1514 * being dropped). If we need to record dependency versioning
1515 * information indicate what version of the needed shared object
1516 * this symbol is part of. Flag the symbol as undefined if it
1517 * has not been made available to us.
1518 */
1519 if ((sdp->sd_ref == REF_DYN_NEED) &&
1520     (!(sdp->sd_flags & FLG_SY_REFRSD))) {
1521     sdp->sd_file->ifl_flags |= FLG_IF_DEPREQD;
1522
1523     /*
1524     * Capture that we've bound to a symbol that doesn't
1525     * allow being directly bound to.
1526     */
1527     if (sdp->sd_flags & FLG_SY_NDIR)
1528         of1->ofl_flags1 |= FLG_OF1_NGLBDIR;
1529
1530     if (sdp->sd_file->ifl_vercnt) {
1531         int vndx;
1532         Ver_index *vip;
1533
1534         vndx = sdp->sd_aux->sa_dverndx;
1535         vip = &sdp->sd_file->ifl_verndx[vndx];
1536         if (vip->vi_flags & FLG_VER_AVAIL) {
1537             vip->vi_flags |= FLG_VER_REFER;
1538         } else {

```

```

1540         sym_undef_entry(of1, sdp, NOTAVAIL,
1541                       FLG_OF_FATAL, &undef_state);
1542         continue;
1543     }
1544 }
1545 }
1546
1547 /*
1548 * Test that we do not bind to symbol supplied from an implicit
1549 * shared object. If a binding is from a weak reference it can
1550 * be ignored.
1551 */
1552 if (needed && !undeferr && (sdp->sd_flags & FLG_SY_GLOBREF) &&
1553     (sdp->sd_ref == REF_DYN_NEED) &&
1554     (sdp->sd_flags & FLG_SY_NOTAVAIL)) {
1555     sym_undef_entry(of1, sdp, IMPLICIT, needed,
1556                   &undef_state);
1557     if (needed == FLG_OF_FATAL)
1558         continue;
1559 }
1560
1561 /*
1562 * Test that a symbol isn't going to be reduced to local scope
1563 * which actually wants to bind to a shared object - if so it's
1564 * a fatal error.
1565 */
1566 if ((sdp->sd_ref == REF_DYN_NEED) &&
1567     (sdp->sd_flags & (FLG_SY_HIDDEN | FLG_SY_PROTECT))) {
1568     sym_undef_entry(of1, sdp, BNDLOCAL, FLG_OF_FATAL,
1569                   &undef_state);
1570     continue;
1571 }
1572
1573 /*
1574 * If the output image is to be versioned then all symbol
1575 * definitions must be associated with a version. Remove any
1576 * versioning that might be left associated with an undefined
1577 * symbol.
1578 */
1579 if (verdesc && (sdp->sd_ref == REF_REL_NEED)) {
1580     if (sym->st_shndx == SHN_UNDEF) {
1581         if (sdp->sd_aux && sdp->sd_aux->sa_overndx)
1582             sdp->sd_aux->sa_overndx = 0;
1583     } else {
1584         if (!SYM_IS_HIDDEN(sdp) && sdp->sd_aux &&
1585             (sdp->sd_aux->sa_overndx == 0)) {
1586             sym_undef_entry(of1, sdp, NOVERSION,
1587                           verdesc, &undef_state);
1588             continue;
1589         }
1590     }
1591 }
1592
1593 /*
1594 * If we don't need the symbol there's no need to process it
1595 * any further.
1596 */
1597 if (sdp->sd_ref == REF_DYN_SEEN)
1598     continue;
1599
1600 /*
1601 * Calculate the size and alignment requirements for the global
1602 * .bss and .tls sections. If we're building a relocatable
1603 * object only account for scoped COMMON symbols (these will
1604 * be converted to .bss references).
1605 */

```

```

1606     * When -z nopartial is in effect, partially initialized
1607     * symbols are directed to the special .data section
1608     * created for that purpose (ofl->ofl_isparexpn).
1609     * Otherwise, partially initialized symbols go to .bss.
1610     *
1611     * Also refer to make_mvsections() in sunwmove.c
1612     */
1613     if ((sym->st_shndx == SHN_COMMON) &&
1614         (((oflags & FLG_OF_RELOBJ) == 0) ||
1615          (SYM_IS_HIDDEN(sdp) && (oflags & FLG_OF_PROCCRED)))) {
1616         if ((sdp->sd_move == NULL) ||
1617             ((sdp->sd_flags & FLG_SY_PAREXP) == 0)) {
1618             if (type != STT_TLS) {
1619                 need_bss = TRUE;
1620                 bsssize = (Xword)S_ROUND(bsssize,
1621                                         sym->st_value) + sym->st_size;
1622                 if (sym->st_value > bssalign)
1623                     bssalign = sym->st_value;
1624             } else {
1625                 need_tlsbss = TRUE;
1626                 tlssize = (Xword)S_ROUND(tlssize,
1627                                         sym->st_value) + sym->st_size;
1628                 if (sym->st_value > tlsalign)
1629                     tlsalign = sym->st_value;
1630             }
1631         }
1632     }
1633
1634 #if defined(_ELF64)
1635     /*
1636     * Calculate the size and alignment requirement for the global
1637     * .lbss. TLS or partially initialized symbols do not need to be
1638     * considered yet.
1639     */
1640     if ((ld_targ.t_m.m_mach == EM_AMD64) &&
1641         (sym->st_shndx == SHN_X86_64_LCOMMON)) {
1642         need_lbss = TRUE;
1643         lbsssize = (Xword)S_ROUND(lbsssize, sym->st_value) +
1644                 sym->st_size;
1645         if (sym->st_value > lbssalign)
1646             lbssalign = sym->st_value;
1647     }
1648 #endif
1649     /*
1650     * If a symbol was referenced via the command line
1651     * (ld -u <>, ...), then this counts as a reference against the
1652     * symbol. Mark any section that symbol is defined in.
1653     */
1654     if (((isp = sdp->sd_isc) != 0) &&
1655         (sdp->sd_flags & FLG_SY_CMDREF)) {
1656         isp->is_flags |= FLG_IS_SECTREF;
1657         isp->is_file->ifl_flags |= FLG_IF_FILEREFF;
1658     }
1659
1660     /*
1661     * Update the symbol count and the associated name string size.
1662     * Note, a capabilities symbol must remain as visible as a
1663     * global symbol. However, the runtime linker recognizes the
1664     * hidden requirement and ensures the symbol isn't made globally
1665     * available at runtime.
1666     */
1667     if (SYM_IS_HIDDEN(sdp) && (oflags & FLG_OF_PROCCRED)) {
1668         /*
1669         * If any reductions are being processed, keep a count
1670         * of eliminated symbols, and if the symbol is being
1671         * reduced to local, count it's size for the .symtab.

```

```

1672     */
1673     if (sdp->sd_flags & FLG_SY_ELIM) {
1674         ofl->ofl_elimcnt++;
1675     } else {
1676         ofl->ofl_scopecnt++;
1677         if (((sdp->sd_flags & FLG_SY_REGSYM) == 0) ||
1678             (sym->st_name && (st_insert(ofl->ofl_strtab,
1679                                     sdp->sd_name) == -1)))
1680             return (S_ERROR);
1681         if (allow_ldynsym && sym->st_name &&
1682             ldynsym_syntype[type]) {
1683             ofl->ofl_dynscopecnt++;
1684             if (st_insert(ofl->ofl_dynstrtab,
1685                           sdp->sd_name) == -1)
1686                 return (S_ERROR);
1687             /* Include it in sort section? */
1688             DYN SORT_COUNT(sdp, sym, type, ++);
1689         }
1690     }
1691 } else {
1692     ofl->ofl_globcnt++;
1693
1694     /*
1695     * Check to see if this global variable should go into
1696     * a sort section. Sort sections require a
1697     * .SUNW_ldynsym section, so, don't check unless a
1698     * .SUNW_ldynsym is allowed.
1699     */
1700     if (allow_ldynsym)
1701         DYN SORT_COUNT(sdp, sym, type, ++);
1702
1703     /*
1704     * If global direct bindings are in effect, or this
1705     * symbol has bound to a dependency which was specified
1706     * as requiring direct bindings, and it hasn't
1707     * explicitly been defined as a non-direct binding
1708     * symbol, mark it.
1709     */
1710     if (((ofl->ofl_dtfldgs_1 & DF_1_DIRECT) || (isp &&
1711         (isp->is_file->ifl_flags & FLG_IF_DIRECT))) &&
1712         ((sdp->sd_flags & FLG_SY_NDIR) == 0))
1713         sdp->sd_flags |= FLG_SY_DIR;
1714
1715     /*
1716     * Insert the symbol name.
1717     */
1718     if (((sdp->sd_flags & FLG_SY_REGSYM) == 0) ||
1719         (sym->st_name) {
1720         if (st_insert(ofl->ofl_strtab,
1721                       sdp->sd_name) == -1)
1722             return (S_ERROR);
1723
1724         if (!(ofl->ofl_flags & FLG_OF_RELOBJ) &&
1725             (st_insert(ofl->ofl_dynstrtab,
1726                       sdp->sd_name) == -1))
1727             return (S_ERROR);
1728     }
1729
1730     /*
1731     * If this section offers a global symbol - record that
1732     * fact.
1733     */
1734     if (isp) {
1735         isp->is_flags |= FLG_IS_SECTREF;
1736         isp->is_file->ifl_flags |= FLG_IF_FILEREFF;
1737     }

```

```

1738     }
1739 }
1741 /*
1742  * Guidance: Use -z defs|nodefs when building shared objects.
1743  *
1744  * Our caller issues this, unless we mask it out here. So we mask it
1745  * out unless we've issued at least one warnings or fatal error.
1746  */
1747 if (!((oflags & FLG_OF_SHAROBJ) && OFL_GUIDANCE(ofl, FLG_OFG_NO_DEFS) &&
1748      (undef_state & (FLG_OF_FATAL | FLG_OF_WARN))))
1749     ofl->ofl_guideflags |= FLG_OFG_NO_DEFS;
1751 /*
1752  * If we've encountered a fatal error during symbol validation then
1753  * return now.
1754  */
1755 if (ofl->ofl_flags & FLG_OF_FATAL)
1756     return (1);
1758 /*
1759  * Now that symbol resolution is completed, scan any register symbols.
1760  * From now on, we're only interested in those that contribute to the
1761  * output file.
1762  */
1763 if (ofl->ofl_regsyms) {
1764     int ndx;
1766     for (ndx = 0; ndx < ofl->ofl_regsymsno; ndx++) {
1767         if ((sdp = ofl->ofl_regsyms[ndx]) == NULL)
1768             continue;
1769         if (sdp->sd_ref != REF_REL_NEED) {
1770             ofl->ofl_regsyms[ndx] = NULL;
1771             continue;
1772         }
1774         ofl->ofl_regsymcnt++;
1775         if (sdp->sd_sym->st_name == 0)
1776             sdp->sd_name = MSG_ORIG(MSG_STR_EMPTY);
1778         if (SYM_IS_HIDDEN(sdp) ||
1779             (ELF_ST_BIND(sdp->sd_sym->st_info) == STB_LOCAL))
1780             ofl->ofl_lregsymcnt++;
1781     }
1782 }
1784 /*
1785  * Generate the .bss section now that we know its size and alignment.
1786  */
1787 if (need_bss) {
1788     if (ld_make_bss(ofl, bsssize, bssalign,
1789                    ld_targ.t_id.id_bss) == S_ERROR)
1790         return (S_ERROR);
1791 }
1792 if (need_tlsbss) {
1793     if (ld_make_bss(ofl, tlbsssize, tlbssalign,
1794                    ld_targ.t_id.id_tlsbss) == S_ERROR)
1795         return (S_ERROR);
1796 }
1797 #if defined(_ELF64)
1798 if ((ld_targ.t_m.m_mach == EM_AMD64) &&
1799     need_lbss && !(oflags & FLG_OF_RELOBJ)) {
1800     if (ld_make_bss(ofl, lbsssize, lbssalign,
1801                    ld_targ.t_id.id_lbss) == S_ERROR)
1802         return (S_ERROR);
1803 }

```

```

1804 #endif
1805 /*
1806  * Determine what entry point symbol we need, and if found save its
1807  * symbol descriptor so that we can update the ELF header entry with the
1808  * symbol's value later (see update_oehdr). Make sure the symbol is
1809  * tagged to ensure its update in case -s is in effect. Use any -e
1810  * option first, or the default entry points '_start' and 'main'.
1811  */
1812 ret = 0;
1813 if (ofl->ofl_entry) {
1814     if ((sdp = ld_sym_find(ofl->ofl_entry, SYM_NOHASH,
1815                          NULL, ofl)) == NULL) {
1816         ld_eprintf(ofl, ERR_FATAL, MSG_INTL(MSG_ARG_NOENTRY),
1817                  ofl->ofl_entry);
1818         ret++;
1819     } else if (ensure_sym_local(ofl, sdp,
1820                               MSG_INTL(MSG_SYM_ENTRY)) != 0) {
1821         ret++;
1822     } else {
1823         ofl->ofl_entry = (void *)sdp;
1824     }
1825 } else if ((sdp = ld_sym_find(MSG_ORIG(MSG_SYM_START),
1826                              SYM_NOHASH, NULL, ofl)) != NULL) && (ensure_sym_local(ofl,
1827                                          sdp, 0) == 0)) {
1828     ofl->ofl_entry = (void *)sdp;
1830 } else if ((sdp = ld_sym_find(MSG_ORIG(MSG_SYM_MAIN),
1831                              SYM_NOHASH, NULL, ofl)) != NULL) && (ensure_sym_local(ofl,
1832                                          sdp, 0) == 0)) {
1833     ofl->ofl_entry = (void *)sdp;
1834 }
1836 /*
1837  * If ld -zdttrace=<sym> was given, then validate that the symbol is
1838  * defined within the current object being built.
1839  */
1840 if ((sdp = ofl->ofl_dtracesym) != 0)
1841     ret += ensure_sym_local(ofl, sdp, MSG_ORIG(MSG_STR_DTRACE));
1843 /*
1844  * If any inittarray, finiarray or preinitarray functions have been
1845  * requested, make sure they are defined within the current object
1846  * being built.
1847  */
1848 if (ofl->ofl_inittarray) {
1849     ret += ensure_array_local(ofl, ofl->ofl_inittarray,
1850                             MSG_ORIG(MSG_SYM_INITARRAY));
1851 }
1852 if (ofl->ofl_finiarray) {
1853     ret += ensure_array_local(ofl, ofl->ofl_finiarray,
1854                             MSG_ORIG(MSG_SYM_FINIARRAY));
1855 }
1856 if (ofl->ofl_preiarray) {
1857     ret += ensure_array_local(ofl, ofl->ofl_preiarray,
1858                             MSG_ORIG(MSG_SYM_PREINITARRAY));
1859 }
1861 if (ret)
1862     return (S_ERROR);
1864 /*
1865  * If we're required to record any needed dependencies versioning
1866  * information calculate it now that all symbols have been validated.
1867  */
1868 if ((oflags & (FLG_OF_VERNEED | FLG_OF_NOVERSEC)) == FLG_OF_VERNEED)
1869     return (ld_vers_check_need(ofl));

```

```

1870     else
1871         return (1);
1872 }

1874 /*
1875  * qsort(3c) comparison function. As an optimization for associating weak
1876  * symbols to their strong counterparts sort global symbols according to their
1877  * section index, address and binding.
1878  */
1879 static int
1880 compare(const void *sdp1, const void *sdp2)
1881 {
1882     Sym_desc    *sdp1 = *((Sym_desc **)sdp1);
1883     Sym_desc    *sdp2 = *((Sym_desc **)sdp2);
1884     Sym         *sym1, *sym2;
1885     uchar_t     bind1, bind2;

1887     /*
1888      * Symbol descriptors may be zero, move these to the front of the
1889      * sorted array.
1890      */
1891     if (sdp1 == NULL)
1892         return (-1);
1893     if (sdp2 == NULL)
1894         return (1);

1896     sym1 = sdp1->sd_sym;
1897     sym2 = sdp2->sd_sym;

1899     /*
1900      * Compare the symbols section index. This is important when sorting
1901      * the symbol tables of relocatable objects. In this case, a symbols
1902      * value is the offset within the associated section, and thus many
1903      * symbols can have the same value, but are effectively different
1904      * addresses.
1905      */
1906     if (sym1->st_shndx > sym2->st_shndx)
1907         return (1);
1908     if (sym1->st_shndx < sym2->st_shndx)
1909         return (-1);

1911     /*
1912      * Compare the symbols value (address).
1913      */
1914     if (sym1->st_value > sym2->st_value)
1915         return (1);
1916     if (sym1->st_value < sym2->st_value)
1917         return (-1);

1919     bind1 = ELF_ST_BIND(sym1->st_info);
1920     bind2 = ELF_ST_BIND(sym2->st_info);

1922     /*
1923      * If two symbols have the same address place the weak symbol before
1924      * any strong counterpart.
1925      */
1926     if (bind1 > bind2)
1927         return (-1);
1928     if (bind1 < bind2)
1929         return (1);

1931     return (0);
1932 }

1934 /*
1935  * Issue a MSG_SYM_BADADDR error from ld_sym_process(). This error

```

```

1936  * is issued when a symbol address/size is not contained by the
1937  * target section.
1938  */
1939  * Such objects are at least partially corrupt, and the user would
1940  * be well advised to be skeptical of them, and to ask their compiler
1941  * supplier to fix the problem. However, a distinction needs to be
1942  * made between symbols that reference readonly text, and those that
1943  * access writable data. Other than throwing off profiling results,
1944  * the readonly section case is less serious. We have encountered
1945  * such objects in the field. In order to allow existing objects
1946  * to continue working, we issue a warning rather than a fatal error
1947  * if the symbol is against readonly text. Other cases are fatal.
1948  */
1949 static void
1950 issue_badaddr_msg(If1_desc *if1, Of1_desc *of1, Sym_desc *sdp,
1951                 Sym *sym, Word shndx)
1952 {
1953     Error      err;
1954     const char *msg;

1956     if ((sdp->sd_isc->is_shdr->sh_flags & (SHF_WRITE | SHF_ALLOC)) ==
1957         SHF_ALLOC) {
1958         msg = MSG_INTL(MSG_SYM_BADADDR_ROTXT);
1959         err = ERR_WARNING;
1960     } else {
1961         msg = MSG_INTL(MSG_SYM_BADADDR);
1962         err = ERR_FATAL;
1963     }

1965     ld_eprintf(of1, err, msg, demangle(sdp->sd_name),
1966              if1->if1_name, shndx, sdp->sd_isc->is_name,
1967              EC_XWORD(sdp->sd_isc->is_shdr->sh_size),
1968              EC_XWORD(sym->st_value), EC_XWORD(sym->st_size));
1969 }

1971 /*
1972  * Global symbols that are candidates for translation to local capability
1973  * symbols under -z symbolcap, are maintained on a local symbol list. Once
1974  * all symbols of a file are processed, this list is traversed to cull any
1975  * unnecessary weak symbol aliases.
1976  */
1977 typedef struct {
1978     Sym_desc    *c_nsd;      /* new lead symbol */
1979     Sym_desc    *c_osdp;    /* original symbol */
1980     Cap_group   *c_group;   /* symbol capability group */
1981     Word        c_ndx;      /* symbol index */
1982 } Cap_pair;

1984 /*
1985  * Process the symbol table for the specified input file. At this point all
1986  * input sections from this input file have been assigned an input section
1987  * descriptor which is saved in the 'ifl_isdesc' array.
1988  *
1989  * - local symbols are saved (as is) if the input file is a relocatable
1990  *   object
1991  *
1992  * - global symbols are added to the linker's internal symbol table if they
1993  *   are not already present, otherwise a symbol resolution function is
1994  *   called upon to resolve the conflict.
1995  */
1996 uintptr_t
1997 ld_sym_process(Is_desc *isc, If1_desc *if1, Of1_desc *of1)
1998 {
1999     /*
2000      * This macro tests the given symbol to see if it is out of
2001      * range relative to the section it references.

```

```

2002      *
2003      * entry:
2004      *   - ifl is a relative object (ET_REL)
2005      *   _sdp - Symbol descriptor
2006      *   _sym - Symbol
2007      *   _type - Symbol type
2008      *
2009      * The following are tested:
2010      *   - Symbol length is non-zero
2011      *   - Symbol type is a type that references code or data
2012      *   - Referenced section is not 0 (indicates an UNDEF symbol)
2013      *   and is not in the range of special values above SHN_LORESERVE
2014      *   (excluding SHN_XINDEX, which is OK).
2015      *   - We have a valid section header for the target section
2016      *
2017      * If the above are all true, and the symbol position is not
2018      * contained by the target section, this macro evaluates to
2019      * True (1). Otherwise, False(0).
2020      */
2021 #define SYM_LOC_BADADDR(_sdp, _sym, _type) \
2022   (_sym->st_size && dynsymsort_symtype[_type] && \
2023   (_sym->st_shndx != SHN_UNDEF) && \
2024   (( _sym->st_shndx < SHN_LORESERVE) || \
2025    ( _sym->st_shndx == SHN_XINDEX)) && \
2026   (_sdp->sd_isc && _sdp->sd_isc->is_shdr && \
2027   ((_sym->st_value + _sym->st_size) > _sdp->sd_isc->is_shdr->sh_size))
2028
2029 Conv_inv_buf_t  inv_buf;
2030 Sym             *sym = (Sym *)isc->is_indata->d_buf;
2031 Word           *symshndx = NULL;
2032 Shdr           *shdr = isc->is_shdr;
2033 Sym_desc       *sdp;
2034 size_t         strsize;
2035 char           *strs;
2036 uchar_t        type, bind;
2037 Word           ndx, hash, local, total;
2038 uchar_t        osabi = ifl->ifl_ehdr->e_ident[EI_OSABI];
2039 Half           mach = ifl->ifl_ehdr->e_machine;
2040 Half           etype = ifl->ifl_ehdr->e_type;
2041 int            etype_rel;
2042 const char     *symsecname, *strsecname;
2043 Word           symsecndx;
2044 avl_index_t    where;
2045 int            test_gnu_hidden_bit, weak;
2046 Cap_desc       *cdp = NULL;
2047 Alist          *cappairs = NULL;
2048
2049 /*
2050  * Its possible that a file may contain more than one symbol table,
2051  * ie. .dynsym and .symtab in a shared library. Only process the first
2052  * table (here, we assume .dynsym comes before .symtab).
2053  */
2054 if (ifl->ifl_symscnt)
2055     return (1);
2056
2057 if (isc->is_symshndx)
2058     symshndx = isc->is_symshndx->is_indata->d_buf;
2059
2060 DBG_CALL(DBG_syms_process(ofl->ofl_lml, ifl));
2061
2062 symsecndx = isc->is_scnndx;
2063 if (isc->is_name)
2064     symsecname = isc->is_name;
2065 else
2066     symsecname = MSG_ORIG(MSG_STR_EMPTY);

```

```

2068      /*
2069      * From the symbol tables section header information determine which
2070      * strtab table is needed to locate the actual symbol names.
2071      */
2072 if (ifl->ifl_flags & FLG_IF_HSTRTAB) {
2073     ndx = shdr->sh_link;
2074     if ((ndx == 0) || (ndx >= ifl->ifl_shnum)) {
2075         ld_eprintf(ofl, ERR_FATAL,
2076             MSG_INTL(MSG_FILE_INVSHLINK), ifl->ifl_name,
2077             EC_WORD(symsecndx), symsecname, EC_XWORD(ndx));
2078         return (S_ERROR);
2079     }
2080     strsize = ifl->ifl_isdesc[ndx]->is_shdr->sh_size;
2081     strs = ifl->ifl_isdesc[ndx]->is_indata->d_buf;
2082     if (ifl->ifl_isdesc[ndx]->is_name)
2083         strsecname = ifl->ifl_isdesc[ndx]->is_name;
2084     else
2085         strsecname = MSG_ORIG(MSG_STR_EMPTY);
2086 } else {
2087     /*
2088     * There is no string table section in this input file
2089     * although there are symbols in this symbol table section.
2090     * This means that these symbols do not have names.
2091     * Currently, only scratch register symbols are allowed
2092     * not to have names.
2093     */
2094     strsize = 0;
2095     strs = (char *)MSG_ORIG(MSG_STR_EMPTY);
2096     strsecname = MSG_ORIG(MSG_STR_EMPTY);
2097 }
2098
2099 /*
2100  * Determine the number of local symbols together with the total
2101  * number we have to process.
2102  */
2103 total = (Word)(shdr->sh_size / shdr->sh_entsize);
2104 local = shdr->sh_info;
2105
2106 /*
2107  * Allocate a symbol table index array and a local symbol array
2108  * (global symbols are processed and added to the ofl->ofl_symbkt[]
2109  * array). If we are dealing with a relocatable object, allocate the
2110  * local symbol descriptors. If this isn't a relocatable object we
2111  * still have to process any shared object locals to determine if any
2112  * register symbols exist. Although these aren't added to the output
2113  * image, they are used as part of symbol resolution.
2114  */
2115 if ((ifl->ifl_oldndx = libld_malloc((size_t)(total *
2116     sizeof(Sym_desc *))) == NULL)
2117     return (S_ERROR);
2118 etype_rel = (etype == ET_REL);
2119 if (etype_rel && local) {
2120     if ((ifl->ifl_locs =
2121         libld_calloc(sizeof(Sym_desc), local)) == NULL)
2122         return (S_ERROR);
2123     /* LINTED */
2124     ifl->ifl_locscnt = (Word)local;
2125 }
2126 ifl->ifl_symscnt = total;
2127
2128 /*
2129  * If there are local symbols to save add them to the symbol table
2130  * index array.
2131  */
2132 if (local) {
2133     int            allow_ldynsym = OFL_ALLOW_LDYSYM(ofl);

```

```

2134     Sym_desc      *last_file_sdp = NULL;
2135     int           last_file_ndx = 0;

2137     for (sym++, ndx = 1; ndx < local; sym++, ndx++) {
2138         sd_flag_t  sdflags = FLG_SY_CLEAN;
2139         Word       shndx;
2140         const char *name;
2141         Sym_desc   *rsdp;
2142         int        shndx_bad = 0;
2143         int        symtab_enter = 1;

2145         /*
2146          * Determine and validate the associated section index.
2147          */
2148         if (symshndx && (sym->st_shndx == SHN_XINDEX)) {
2149             shndx = symshndx[ndx];
2150         } else if ((shndx = sym->st_shndx) >= SHN_LORESERVE) {
2151             sdflags |= FLG_SY_SPECSEC;
2152         } else if (shndx > ifl->ifl_shnum) {
2153             /* We need the name before we can issue error */
2154             shndx_bad = 1;
2155         }

2157         /*
2158          * Check if st_name has a valid value or not.
2159          */
2160         if ((name = string(ofl, ifl, sym, strsize, ndx,
2161             shndx, symsecndx, symsecname, strsecname,
2162             &sdflags)) == NULL)
2163             continue;

2165         /*
2166          * Now that we have the name, if the section index
2167          * was bad, report it.
2168          */
2169         if (shndx_bad) {
2170             ld_eprintf(ofl, ERR_WARNING,
2171                 MSG_INTL(MSG_SYM_INVSHNDX),
2172                 demangle_symname(name, symsecname, ndx),
2173                 ifl->ifl_name,
2174                 conv_sym_shndx(osabi, mach, sym->st_shndx,
2175                     CONV_FMT_DECIMAL, &inv_buf));
2176             continue;
2177         }

2179         /*
2180          * If this local symbol table originates from a shared
2181          * object, then we're only interested in recording
2182          * register symbols. As local symbol descriptors aren't
2183          * allocated for shared objects, one will be allocated
2184          * to associated with the register symbol. This symbol
2185          * won't become part of the output image, but we must
2186          * process it to test for register conflicts.
2187          */
2188         rsdp = sdp = NULL;
2189         if (sdflags & FLG_SY_REGSYM) {
2190             /*
2191              * The presence of FLG_SY_REGSYM means that
2192              * the pointers in ld_targ.t_ms are non-NULL.
2193              */
2194             rsdp = (*ld_targ.t_ms.ms_reg_find)(sym, ofl);
2195             if (rsdp != 0) {
2196                 /*
2197                  * The fact that another register def-
2198                  * inition has been found is fatal.
2199                  * Call the verification routine to get

```

```

2200             * the error message and move on.
2201             */
2202             (void) (*ld_targ.t_ms.ms_reg_check)
2203                 (rsdp, sym, name, ifl, ofl);
2204             continue;
2205         }

2207         if (etype == ET_DYN) {
2208             if ((sdp = libld_calloc(
2209                 sizeof (Sym_desc), 1)) == NULL)
2210                 return (S_ERROR);
2211             sdp->sd_ref = REF_DYN_SEEN;

2213             /* Will not appear in output object */
2214             symtab_enter = 0;
2215         }
2216     } else if (etype == ET_DYN)
2217         continue;

2219     /*
2220      * Fill in the remaining symbol descriptor information.
2221      */
2222     if (sdp == NULL) {
2223         sdp = &(ifl->ifl_locs[ndx]);
2224         sdp->sd_ref = REF_REL_NEED;
2225         sdp->sd_symndx = ndx;
2226     }
2227     if (rsdp == NULL) {
2228         sdp->sd_name = name;
2229         sdp->sd_sym = sym;
2230         sdp->sd_shndx = shndx;
2231         sdp->sd_flags = sdflags;
2232         sdp->sd_file = ifl;
2233         ifl->ifl_oldndx[ndx] = sdp;
2234     }

2236     DBG_CALL(DBG_syms_entry(ofl->ofl_lml, ndx, sdp));

2238     /*
2239      * Reclassify any SHN_SUNW_IGNORE symbols to SHN_UNDEF
2240      * so as to simplify future processing.
2241      */
2242     if (sym->st_shndx == SHN_SUNW_IGNORE) {
2243         sdp->sd_shndx = shndx = SHN_UNDEF;
2244         sdp->sd_flags |= (FLG_SY_IGNORE | FLG_SY_ELIM);
2245     }

2247     /*
2248      * Process any register symbols.
2249      */
2250     if (sdp->sd_flags & FLG_SY_REGSYM) {
2251         /*
2252          * Add a diagnostic to indicate we've caught a
2253          * register symbol, as this can be useful if a
2254          * register conflict is later discovered.
2255          */
2256         DBG_CALL(DBG_syms_entered(ofl, sym, sdp));

2258     /*
2259      * If this register symbol hasn't already been
2260      * recorded, enter it now.
2261      *
2262      * The presence of FLG_SY_REGSYM means that
2263      * the pointers in ld_targ.t_ms are non-NULL.
2264      */
2265     if ((rsdp == NULL) &&

```

```

2266         ((*ld_targ.t_ms.ms_reg_enter)(sdp, ofl) ==
2267         0))
2268         return (S_ERROR);
2269     }
2271     /*
2272     * Assign an input section.
2273     */
2274     if ((sym->st_shndx != SHN_UNDEF) &&
2275         ((sdp->sd_flags & FLG_SY_SPECSEC) == 0))
2276         sdp->sd_isc = ifl->ifl_iscdesc[shndx];
2278     /*
2279     * If this symbol falls within the range of a section
2280     * being discarded, then discard the symbol itself.
2281     * There is no reason to keep this local symbol.
2282     */
2283     if (sdp->sd_isc &&
2284         (sdp->sd_isc->is_flags & FLG_IS_DISCARD)) {
2285         sdp->sd_flags |= FLG_SY_ISDISC;
2286         DBG_CALL(DBG_syms_discarded(ofl->ofl_lml, sdp));
2287         continue;
2288     }
2290     /*
2291     * Skip any section symbols as new versions of these
2292     * will be created.
2293     */
2294     if ((type = ELF_ST_TYPE(sym->st_info)) == STT_SECTION) {
2295         if (sym->st_shndx == SHN_UNDEF) {
2296             ld_eprintf(ofl, ERR_WARNING,
2297                 MSG_INTL(MSG_SYM_INVSHNDX),
2298                 demangle_symname(name, symsecname,
2299                     ndx), ifl->ifl_name,
2300                 conv_sym_shndx(osabi, mach,
2301                     sym->st_shndx, CONV_FMT_DECIMAL,
2302                     &inv_buf));
2303         }
2304         continue;
2305     }
2307     /*
2308     * For a relocatable object, if this symbol is defined
2309     * and has non-zero length and references an address
2310     * within an associated section, then check its extents
2311     * to make sure the section boundaries encompass it.
2312     * If they don't, the ELF file is corrupt.
2313     */
2314     if (etype_rel) {
2315         if (SYM_LOC_BADADDR(sdp, sym, type)) {
2316             issue_badaddr_msg(ifl, ofl, sdp,
2317                 sym, shndx);
2318             if (ofl->ofl_flags & FLG_OF_FATAL)
2319                 continue;
2320         }
2322     /*
2323     * We have observed relocatable objects
2324     * containing identical adjacent STT_FILE
2325     * symbols. Discard any other than the first,
2326     * as they are all equivalent and the extras
2327     * do not add information.
2328     *
2329     * For the purpose of this test, we assume
2330     * that only the symbol type and the string
2331     * table offset (st_name) matter.

```

```

2332     */
2333     if (type == STT_FILE) {
2334         int toss = (last_file_sdp != NULL) &&
2335             ((ndx - 1) == last_file_ndx) &&
2336             (sym->st_name ==
2337             last_file_sdp->sd_sym->st_name);
2339         last_file_sdp = sdp;
2340         last_file_ndx = ndx;
2341         if (toss) {
2342             sdp->sd_flags |= FLG_SY_INVALID;
2343             DBG_CALL(DBG_syms_dup_discarded(
2344                 ofl->ofl_lml, ndx, sdp));
2345             continue;
2346         }
2347     }
2348 }
2351     /*
2352     * Sanity check for TLS
2353     */
2354     if ((sym->st_size != 0) && ((type == STT_TLS) &&
2355         (sym->st_shndx != SHN_COMMON))) {
2356         Is_desc *isp = sdp->sd_isc;
2358         if ((isp == NULL) || (isp->is_shdr == NULL) ||
2359             ((isp->is_shdr->sh_flags & SHF_TLS) == 0)) {
2360             ld_eprintf(ofl, ERR_FATAL,
2361                 MSG_INTL(MSG_SYM_TLS),
2362                 demangle(sdp->sd_name),
2363                 ifl->ifl_name);
2364             continue;
2365         }
2366     }
2368     /*
2369     * Carry our some basic sanity checks (these are just
2370     * some of the erroneous symbol entries we've come
2371     * across, there's probably a lot more). The symbol
2372     * will not be carried forward to the output file, which
2373     * won't be a problem unless a relocation is required
2374     * against it.
2375     */
2376     if (((sdp->sd_flags & FLG_SY_SPECSEC) &&
2377         ((sym->st_shndx == SHN_COMMON) ||
2378         (type == STT_FILE) &&
2379         (sym->st_shndx != SHN_ABS))) ||
2380         (sdp->sd_isc && (sdp->sd_isc->is_osdesc == NULL))) {
2381         ld_eprintf(ofl, ERR_WARNING,
2382             MSG_INTL(MSG_SYM_INVSHNDX),
2383             demangle_symname(name, symsecname, ndx),
2384             ifl->ifl_name,
2385             conv_sym_shndx(osabi, mach, sym->st_shndx,
2386                 CONV_FMT_DECIMAL, &inv_buf));
2387         sdp->sd_isc = NULL;
2388         sdp->sd_flags |= FLG_SY_INVALID;
2389         continue;
2390     }
2392     /*
2393     * As these local symbols will become part of the output
2394     * image, record their number and name string size.
2395     * Globals are counted after all input file processing
2396     * (and hence symbol resolution) is complete during
2397     * sym_validate().

```



```

2398      */
2399      if (!(ofl->ofl_flags & FLG_OF_REDLSYM) &&
2400          symtab_enter) {
2401          ofl->ofl_locscnt++;
2402
2403          if (((sdp->sd_flags & FLG_SY_REGSYM) == 0) ||
2404              sym->st_name) && (st_insert(ofl->ofl_strtab,
2405              sdp->sd_name) == -1))
2406              return (S_ERROR);
2407
2408          if (allow_ldynsym && sym->st_name &&
2409              ldynsym_syntype[type]) {
2410              ofl->ofl_dynlocsnt++;
2411              if (st_insert(ofl->ofl_dynstrtab,
2412                  sdp->sd_name) == -1)
2413                  return (S_ERROR);
2414              /* Include it in sort section? */
2415              DYN_SORT_COUNT(sdp, sym, type, ++);
2416          }
2417      }
2418  }
2419  }
2420
2421  /*
2422  * The GNU ld interprets the top bit of the 16-bit Versym value
2423  * (0x8000) as the "hidden" bit. If this bit is set, the linker
2424  * is supposed to act as if that symbol does not exist. The Solaris
2425  * linker does not support this mechanism, or the model of interface
2426  * evolution that it allows, but we honor it in GNU ld produced
2427  * objects in order to interoperate with them.
2428  *
2429  * Determine if we should honor the GNU hidden bit for this file.
2430  */
2431  test_gnu_hidden_bit = ((ifl->ifl_flags & FLG_IF_GNUVER) != 0) &&
2432      (ifl->ifl_versym != NULL);
2433
2434  /*
2435  * Determine whether object capabilities for this file are being
2436  * converted into symbol capabilities. If so, global function symbols,
2437  * and initialized global data symbols, need special translation and
2438  * processing.
2439  */
2440  if ((etype == ET_REL) && (ifl->ifl_flags & FLG_IF_OTOSCAP))
2441      cdp = ifl->ifl_caps;
2442
2443  /*
2444  * Now scan the global symbols entering them in the internal symbol
2445  * table or resolving them as necessary.
2446  */
2447  sym = (Sym *)isc->is_indata->d_buf;
2448  sym += local;
2449  weak = 0;
2450  /* LINTED */
2451  for (ndx = (int)local; ndx < total; sym++, ndx++) {
2452      const char      *name;
2453      sd_flag_t      sdflags = 0;
2454      Word           shndx;
2455      int            shndx_bad = 0;
2456      Sym            *nsym = sym;
2457      Cap_pair       *cpp = NULL;
2458      uchar_t        ntype;
2459
2460      /*
2461      * Determine and validate the associated section index.
2462      */
2463      if (symshndx && (nsym->st_shndx == SHN_XINDEX)) {

```

```

2464          shndx = symshndx[ndx];
2465      } else if ((shndx = nsym->st_shndx) >= SHN_LORESERVE) {
2466          sdflags |= FLG_SY_SPECSEC;
2467      } else if (shndx > ifl->ifl_shnum) {
2468          /* We need the name before we can issue error */
2469          shndx_bad = 1;
2470      }
2471
2472      /*
2473      * Check if st_name has a valid value or not.
2474      */
2475      if ((name = string(ofl, ifl, nsym, str, strsize, ndx, shndx,
2476          symsecndx, symsecname, strsecname, &sdflags)) == NULL)
2477          continue;
2478
2479      /*
2480      * Now that we have the name, report an erroneous section index.
2481      */
2482      if (shndx_bad) {
2483          ld_eprintf(ofl, ERR_WARNING, MSG_INTL(MSG_SYM_INVSHNDX),
2484              demangle_symname(name, symsecname, ndx),
2485              ifl->ifl_name,
2486              conv_sym_shndx(osabi, mach, nsym->st_shndx,
2487                  CONV_FMT_DECIMAL, &inv_buf));
2488          continue;
2489      }
2490
2491      /*
2492      * Test for the GNU hidden bit, and ignore symbols that
2493      * have it set.
2494      */
2495      if (test_gnu_hidden_bit &&
2496          ((ifl->ifl_versym[ndx] & 0x8000) != 0))
2497          continue;
2498
2499      /*
2500      * The linker itself will generate symbols for _end, _etext,
2501      * _edata, _DYNAMIC and _PROCEDURE_LINKAGE_TABLE, so don't
2502      * bother entering these symbols from shared objects. This
2503      * results in some wasted resolution processing, which is hard
2504      * to feel, but if nothing else, pollutes diagnostic relocation
2505      * output.
2506      */
2507      if (name[0] && (etype == ET_DYN) && (nsym->st_size == 0) &&
2508          (ELF_ST_TYPE(nsym->st_info) == STT_OBJECT) &&
2509          (name[0] == '_' && ((name[1] == 'e') ||
2510              (name[1] == 'D') || (name[1] == 'P')) &&
2511              ((strcmp(name, MSG_ORIG(MSG_SYM_ETEXT_U)) == 0) ||
2512              (strcmp(name, MSG_ORIG(MSG_SYM_EDATA_U)) == 0) ||
2513              (strcmp(name, MSG_ORIG(MSG_SYM_END_U)) == 0) ||
2514              (strcmp(name, MSG_ORIG(MSG_SYM_DYNAMIC_U)) == 0) ||
2515              (strcmp(name, MSG_ORIG(MSG_SYM_PLKTB_L_U)) == 0))) {
2516          ifl->ifl_oldndx[ndx] = 0;
2517          continue;
2518      }
2519
2520      /*
2521      * The '-z wrap=XXX' option emulates the GNU ld --wrap=XXX
2522      * option. When XXX is the symbol to be wrapped:
2523      *
2524      * - An undefined reference to XXX is converted to __wrap_XXX
2525      * - An undefined reference to __real_XXX is converted to XXX
2526      *
2527      * The idea is that the user can supply a wrapper function
2528      * __wrap_XXX that does some work, and then uses the name
2529      * __real_XXX to pass the call on to the real function. The

```

```

2530     * wrapper objects are linked with the original unmodified
2531     * objects to produce a wrapped version of the output object.
2532     */
2533     if (ofl->ofl_wrap && name[0] && (shndx == SHN_UNDEF)) {
2534         WrapSymNode wsn, *wsnp;

2536         /*
2537          * If this is the __real_XXX form, advance the
2538          * pointer to reference the wrapped name.
2539          */
2540         wsn.wsn_name = name;
2541         if ((*name == '_' &&
2542             (strcmp(name, MSG_ORIG(MSG_STR_UU_REAL_U),
2543                  MSG_STR_UU_REAL_U_SIZE) == 0))
2544             wsn.wsn_name += MSG_STR_UU_REAL_U_SIZE;

2546         /*
2547          * Is this symbol in the wrap AVL tree? If so, map
2548          * XXX to __wrap_XXX, and __real_XXX to XXX. Note that
2549          * wsn.wsn_name will equal the current value of name
2550          * if the __real_prefix is not present.
2551          */
2552         if ((wsnp = avl_find(ofl->ofl_wrap, &wsn, 0)) != NULL) {
2553             const char *old_name = name;

2555             name = (wsn.wsn_name == name) ?
2556                   wsn->wsn_wrapname : wsn.wsn_name;
2557             DBG_CALL(DBG_syms_wrap(ofl->ofl_lml, ndx,
2558                                  old_name, name));
2560         }

2562         /*
2563          * Determine and validate the symbols binding.
2564          */
2565         bind = ELF_ST_BIND(nsym->st_info);
2566         if ((bind != STB_GLOBAL) && (bind != STB_WEAK)) {
2567             ld_eprintf(ofl, ERR_WARNING, MSG_INTL(MSG_SYM_NONGLOB),
2568                      demangle_symname(name, symsecname, ndx),
2569                      ifl->ifl_name,
2570                      conv_sym_info_bind(bind, 0, &inv_buf));
2571             continue;
2572         }
2573         if (bind == STB_WEAK)
2574             weak++;

2576         /*
2577          * If this symbol falls within the range of a section being
2578          * discarded, then discard the symbol itself.
2579          */
2580         if (((sdflags & FLG_SY_SPECSEC) == 0) &&
2581             (nsym->st_shndx != SHN_UNDEF)) {
2582             Is_desc *isp;

2584             if (shndx >= ifl->ifl_shnum) {
2585                 /*
2586                  * Carry our some basic sanity checks
2587                  * The symbol will not be carried forward to
2588                  * the output file, which won't be a problem
2589                  * unless a relocation is required against it.
2590                  */
2591                 ld_eprintf(ofl, ERR_WARNING,
2592                          MSG_INTL(MSG_SYM_INVSHNDX),
2593                          demangle_symname(name, symsecname, ndx),
2594                          ifl->ifl_name,
2595                          conv_sym_shndx(osabi, mach, nsym->st_shndx,

```

```

2596             CONV_FMT_DECIMAL, &inv_buf));
2597             continue;
2598         }

2600         isp = ifl->ifl_isdesc[shndx];
2601         if (isp && (isp->is_flags & FLG_IS_DISCARD)) {
2602             if ((sdp =
2603                 libld_calloc(sizeof (Sym_desc), 1)) == NULL)
2604                 return (S_ERROR);

2606             /*
2607              * Create a dummy symbol entry so that if we
2608              * find any references to this discarded symbol
2609              * we can compensate.
2610              */
2611             sdp->sd_name = name;
2612             sdp->sd_sym = nsym;
2613             sdp->sd_file = ifl;
2614             sdp->sd_isc = isp;
2615             sdp->sd_flags = FLG_SY_ISDISC;
2616             ifl->ifl_oldndx[ndx] = sdp;

2618             DBG_CALL(DBG_syms_discarded(ofl->ofl_lml, sdp));
2619             continue;
2620         }
2621     }

2623     /*
2624     * If object capabilities for this file are being converted
2625     * into symbol capabilities, then:
2626     *
2627     * - Any global function, or initialized global data symbol
2628     *   definitions (ie., those that are not associated with
2629     *   special symbol types, ie., ABS, COMMON, etc.), and which
2630     *   have not been reduced to locals, are converted to symbol
2631     *   references (UNDEF). This ensures that any reference to
2632     *   the original symbol, for example from a relocation, get
2633     *   associated to a capabilities family lead symbol, ie., a
2634     *   generic instance.
2635     *
2636     * - For each global function, or object symbol definition,
2637     *   a new local symbol is created. The function or object
2638     *   is renamed using the capabilities CA_SUNW_ID definition
2639     *   (which might have been fabricated for this purpose -
2640     *   see get_cap_group()). The new symbol name is:
2641     *
2642     *     <original name>%<capability group identifier>
2643     *
2644     * This symbol is associated to the same location, and
2645     * becomes a capabilities family member.
2646     */
2647     /* LINTED */
2648     hash = (Word)elf_hash(name);

2650     ntype = ELF_ST_TYPE(nsym->st_info);
2651     if (cdp && (nsym->st_shndx != SHN_UNDEF) &&
2652         ((sdflags & FLG_SY_SPECSEC) == 0) &&
2653         ((ntype == STT_FUNC) || (ntype == STT_OBJECT))) {
2654         /*
2655          * Determine this symbol's visibility. If a mapfile has
2656          * indicated this symbol should be local, then there's
2657          * no point in transforming this global symbol to a
2658          * capabilities symbol. Otherwise, create a symbol
2659          * capability pair descriptor to record this symbol as
2660          * a candidate for translation.
2661          */

```

```

2662         if (sym_cap_vis(name, hash, sym, ofl) &&
2663             ((cpp = alist_append(&cappairs, NULL,
2664                 sizeof (Cap_pair), AL_CNT_CAP_PAIRS)) == NULL))
2665             return (S_ERROR);
2666     }
2668     if (cpp) {
2669         Sym      *rsym;
2671         DBG_CALL(Dbg_syms_cap_convert(ofl, ndx, name, nsym));
2673         /*
2674          * Allocate a new symbol descriptor to represent the
2675          * transformed global symbol. The descriptor points
2676          * to the original symbol information (which might
2677          * indicate a global or weak visibility). The symbol
2678          * information will be transformed into a local symbol
2679          * later, after any weak aliases are culled.
2680          */
2681         if ((cpp->c_osdp =
2682             libld_malloc(sizeof (Sym_desc)) == NULL)
2683             return (S_ERROR);
2685         cpp->c_osdp->sd_name = name;
2686         cpp->c_osdp->sd_sym = nsym;
2687         cpp->c_osdp->sd_shndx = shndx;
2688         cpp->c_osdp->sd_file = ifl;
2689         cpp->c_osdp->sd_isc = ifl->ifl_iscdesc[shndx];
2690         cpp->c_osdp->sd_ref = REF_REL_NEED;
2692         /*
2693          * Save the capabilities group this symbol belongs to,
2694          * and the original symbol index.
2695          */
2696         cpp->c_group = cdp->ca_groups->apl_data[0];
2697         cpp->c_ndx = ndx;
2699         /*
2700          * Replace the original symbol definition with a symbol
2701          * reference. Make sure this reference isn't left as a
2702          * weak.
2703          */
2704         if ((rsym = libld_malloc(sizeof (Sym))) == NULL)
2705             return (S_ERROR);
2707         *rsym = *nsym;
2709         rsym->st_info = ELF_ST_INFO(STB_GLOBAL, ntype);
2710         rsym->st_shndx = shndx = SHN_UNDEF;
2711         rsym->st_value = 0;
2712         rsym->st_size = 0;
2714         sdflags |= FLG_SY_CAP;
2716         nsym = rsym;
2717     }
2719     /*
2720     * If the symbol does not already exist in the internal symbol
2721     * table add it, otherwise resolve the conflict. If the symbol
2722     * from this file is kept, retain its symbol table index for
2723     * possible use in associating a global alias.
2724     */
2725     if ((sdp = ld_sym_find(name, hash, &where, ofl)) == NULL) {
2726         DBG_CALL(Dbg_syms_global(ofl->ofl_lml, ndx, name));
2727         if ((sdp = ld_sym_enter(name, nsym, hash, ifl, ofl, ndx,

```

```

2728             shndx, sdflags, &where)) == (Sym_desc *)S_ERROR)
2729             return (S_ERROR);
2731     } else if (ld_sym_resolve(sdp, nsym, ifl, ofl, ndx, shndx,
2732         sdflags) == S_ERROR)
2733         return (S_ERROR);
2735     /*
2736     * Now that we have a symbol descriptor, retain the descriptor
2737     * for later use by symbol capabilities processing.
2738     */
2739     if (cpp)
2740         cpp->c_nsdp = sdp;
2742     /*
2743     * After we've compared a defined symbol in one shared
2744     * object, flag the symbol so we don't compare it again.
2745     */
2746     if ((etype == ET_DYN) && (nsym->st_shndx != SHN_UNDEF) &&
2747         ((sdp->sd_flags & FLG_SY_SOFOUND) == 0))
2748         sdp->sd_flags |= FLG_SY_SOFOUND;
2750     /*
2751     * If the symbol is accepted from this file retain the symbol
2752     * index for possible use in aliasing.
2753     */
2754     if (sdp->sd_file == ifl)
2755         sdp->sd_symndx = ndx;
2757     ifl->ifl_oldndx[ndx] = sdp;
2759     /*
2760     * If we've accepted a register symbol, continue to validate
2761     * it.
2762     */
2763     if (sdp->sd_flags & FLG_SY_REGSYM) {
2764         Sym_desc      *rsdp;
2766         /*
2767         * The presence of FLG_SY_REGSYM means that
2768         * the pointers in ld_targ.t_ms are non-NULL.
2769         */
2770         rsdp = (*ld_targ.t_ms.ms_reg_find)(sdp->sd_sym, ofl);
2771         if (rsdp == NULL) {
2772             if ((*ld_targ.t_ms.ms_reg_enter)(sdp, ofl) == 0)
2773                 return (S_ERROR);
2774             } else if (rsdp != sdp) {
2775                 (void) (*ld_targ.t_ms.ms_reg_check)(rsdp,
2776                     sdp->sd_sym, sdp->sd_name, ifl, ofl);
2777             }
2778         }
2780     /*
2781     * For a relocatable object, if this symbol is defined
2782     * and has non-zero length and references an address
2783     * within an associated section, then check its extents
2784     * to make sure the section boundaries encompass it.
2785     * If they don't, the ELF file is corrupt. Note that this
2786     * global symbol may have come from another file to satisfy
2787     * an UNDEF symbol of the same name from this one. In that
2788     * case, we don't check it, because it was already checked
2789     * as part of its own file.
2790     */
2791     if (etype_rel && (sdp->sd_file == ifl)) {
2792         Sym *tsym = sdp->sd_sym;

```

```

2794         if (SYM_LOC_BADADDR(sdp, tsym,
2795             ELF_ST_TYPE(tsym->st_info)) {
2796             issue_badaddr_msg(If1, of1, sdp,
2797                 tsym, tsym->st_shndx);
2798             continue;
2799         }
2800     }
2801 }
2802 DBG_CALL(DBG_util_nl(of1->of1_lml, DBG_NL_STD));

2804 /*
2805 * Associate weak (alias) symbols to their non-weak counterparts by
2806 * scanning the global symbols one more time.
2807 *
2808 * This association is needed when processing the symbols from a shared
2809 * object dependency when a weak definition satisfies a reference:
2810 *
2811 * - When building a dynamic executable, if a referenced symbol is a
2812 *   data item, the symbol data is copied to the executables address
2813 *   space. In this copy-relocation case, we must also reassociate
2814 *   the alias symbol with its new location in the executable.
2815 *
2816 * - If the referenced symbol is a function then we may need to
2817 *   promote the symbols binding from undefined weak to undefined,
2818 *   otherwise the run-time linker will not generate the correct
2819 *   relocation error should the symbol not be found.
2820 *
2821 * Weak alias association is also required when a local dynsym table
2822 * is being created. This table should only contain one instance of a
2823 * symbol that is associated to a given address.
2824 *
2825 * The true association between a weak/strong symbol pair is that both
2826 * symbol entries are identical, thus first we create a sorted symbol
2827 * list keyed off of the symbols section index and value. If the symbol
2828 * belongs to the same section and has the same value, then the chances
2829 * are that the rest of the symbols data is the same. This list is then
2830 * scanned for weak symbols, and if one is found then any strong
2831 * association will exist in the entries that follow. Thus we just have
2832 * to scan one (typically a single alias) or more (in the uncommon
2833 * instance of multiple weak to strong associations) entries to
2834 * determine if a match exists.
2835 */
2836 if (weak && (OFL_ALLOW_LDYNYSYM(of1) || (etype == ET_DYN)) &&
2837     (total > local)) {
2838     static Sym_desc **sort;
2839     static size_t osize = 0;
2840     size_t nsize = (total - local) * sizeof (Sym_desc *);

2842     /*
2843     * As we might be processing many input files, and many symbols,
2844     * try and reuse a static sort buffer. Note, presently we're
2845     * playing the game of never freeing any buffers as there's a
2846     * belief this wastes time.
2847     */
2848     if ((osize == 0) || (nsize > osize)) {
2849         if ((sort = libld_malloc(nsize)) == NULL)
2850             return (S_ERROR);
2851         osize = nsize;
2852     }
2853     (void) memcpy((void *)sort, &if1->if1_olddndx[local], nsize);

2855     qsort(sort, (total - local), sizeof (Sym_desc *), compare);

2857     for (ndx = 0; ndx < (total - local); ndx++) {
2858         Sym_desc *wsdp = sort[ndx];
2859         Sym *wsym;

```

```

2860         int sndx;

2862         /*
2863         * Ignore any empty symbol descriptor, or the case where
2864         * the symbol has been resolved to a different file.
2865         */
2866         if ((wsdp == NULL) || (wsdp->sd_file != if1))
2867             continue;

2869         wsym = wsdp->sd_sym;

2871         if ((wsym->st_shndx == SHN_UNDEF) ||
2872             (wsdp->sd_flags & FLG_SY_SPECSEC) ||
2873             (ELF_ST_BIND(wsym->st_info) != STB_WEAK))
2874             continue;

2876         /*
2877         * We have a weak symbol, if it has a strong alias it
2878         * will have been sorted to one of the following sort
2879         * table entries. Note that we could have multiple weak
2880         * symbols aliased to one strong (if this occurs then
2881         * the strong symbol only maintains one alias back to
2882         * the last weak).
2883         */
2884         for (sndx = ndx + 1; sndx < (total - local); sndx++) {
2885             Sym_desc *ssdp = sort[sndx];
2886             Sym *ssym;
2887             sd_flag_t w_dynbits, s_dynbits;

2889             /*
2890             * Ignore any empty symbol descriptor, or the
2891             * case where the symbol has been resolved to a
2892             * different file.
2893             */
2894             if ((ssdp == NULL) || (ssdp->sd_file != if1))
2895                 continue;

2897             ssym = ssdp->sd_sym;

2899             if (ssym->st_shndx == SHN_UNDEF)
2900                 continue;

2902             if ((ssym->st_shndx != wsym->st_shndx) ||
2903                 (ssym->st_value != wsym->st_value))
2904                 break;

2906             if ((ssym->st_size != wsym->st_size) ||
2907                 (ssdp->sd_flags & FLG_SY_SPECSEC) ||
2908                 (ELF_ST_BIND(ssym->st_info) == STB_WEAK))
2909                 continue;

2911             /*
2912             * If a sharable object, set link fields so
2913             * that they reference each other.
2914             */
2915             if (etype == ET_DYN) {
2916                 ssdp->sd_aux->sa_linkndx =
2917                     (Word)wsdp->sd_symndx;
2918                 wsdp->sd_aux->sa_linkndx =
2919                     (Word)ssdp->sd_symndx;
2920             }

2922             /*
2923             * Determine which of these two symbols go into
2924             * the sort section. If a mapfile has made
2925             * explicit settings of the FLG_SY_*DYNYSORT

```

```

2926     * flags for both symbols, then we do what they
2927     * say.  If one has the DYNBORT flags set, we
2928     * set the NODYNSORT bit in the other.  And if
2929     * neither has an explicit setting, then we
2930     * favor the weak symbol because they usually
2931     * lack the leading underscore.
2932     */
2933     w_dynbits = wsdp->sd_flags &
2934     (FLG_SY_DYNSORT | FLG_SY_NODYNSORT);
2935     s_dynbits = ssdp->sd_flags &
2936     (FLG_SY_DYNSORT | FLG_SY_NODYNSORT);
2937     if (!(w_dynbits && s_dynbits)) {
2938         if (s_dynbits) {
2939             if (s_dynbits == FLG_SY_DYNSORT)
2940                 wsdp->sd_flags |=
2941                     FLG_SY_NODYNSORT;
2942             } else if (w_dynbits !=
2943                 FLG_SY_NODYNSORT) {
2944                 ssdp->sd_flags |=
2945                     FLG_SY_NODYNSORT;
2946             }
2947         }
2948         break;
2949     }
2950 }
2951
2953 /*
2954 * Having processed all symbols, under -z symbolcap, reprocess any
2955 * symbols that are being translated from global to locals.  The symbol
2956 * pair that has been collected defines the original symbol (c_osdp),
2957 * which will become a local, and the new symbol (c_nsdp), which will
2958 * become a reference (UNDEF) for the original.
2959 *
2960 * Scan these symbol pairs looking for weak symbols, which have non-weak
2961 * aliases.  There is no need to translate both of these symbols to
2962 * locals, only the global is necessary.
2963 */
2964 if (cappairs) {
2965     Aliste      idx1;
2966     Cap_pair    *cppl;
2967
2968     for (ALIST_TRAVERSE(cappairs, idx1, cppl)) {
2969         Sym_desc *sdp1 = cppl->c_osdp;
2970         Sym       *sym1 = sdp1->sd_sym;
2971         uchar_t   bind1 = ELF_ST_BIND(sym1->st_info);
2972         Aliste    idx2;
2973         Cap_pair  *cpp2;
2974
2975         /*
2976          * If this symbol isn't weak, it's capability member is
2977          * retained for the creation of a local symbol.
2978          */
2979         if (bind1 != STB_WEAK)
2980             continue;
2981
2982         /*
2983          * If this is a weak symbol, traverse the capabilities
2984          * list again to determine if a corresponding non-weak
2985          * symbol exists.
2986          */
2987         for (ALIST_TRAVERSE(cappairs, idx2, cpp2)) {
2988             Sym_desc *sdp2 = cpp2->c_osdp;
2989             Sym       *sym2 = sdp2->sd_sym;
2990             uchar_t   bind2 =
2991                 ELF_ST_BIND(sym2->st_info);

```

```

2993         if ((cppl == cpp2) ||
2994             (cppl->c_group != cpp2->c_group) ||
2995             (sym1->st_value != sym2->st_value) ||
2996             (bind2 == STB_WEAK))
2997             continue;
2998
2999         /*
3000          * The weak symbol (sym1) has a non-weak (sym2)
3001          * counterpart.  There's no point in translating
3002          * both of these equivalent symbols to locals.
3003          * Add this symbol capability alias to the
3004          * capabilities family information, and remove
3005          * the weak symbol.
3006          */
3007         if (ld_cap_add_family(ofl, cpp2->c_nsdp,
3008             cppl->c_nsdp, NULL, NULL) == S_ERROR)
3009             return (S_ERROR);
3010
3011         free((void *)cppl->c_osdp);
3012         (void) alist_delete(cappairs, &idx1);
3013     }
3014 }
3015
3016 DBG_CALL(DBG_util_nl(ofl->ofl_lml, DBG_NL_STD));
3017
3018 /*
3019 * The capability pairs information now represents all the
3020 * global symbols that need transforming to locals.  These
3021 * local symbols are renamed using their group identifiers.
3022 */
3023 for (ALIST_TRAVERSE(cappairs, idx1, cppl)) {
3024     Sym_desc *osdp = cppl->c_osdp;
3025     Objcset  *capset;
3026     size_t   nsize, tsize;
3027     const char *oname;
3028     char      *cname, *idstr;
3029     Sym       *csym;
3030
3031     /*
3032      * If the local symbol has not yet been translated
3033      * convert it to a local symbol with a name.
3034      */
3035     if ((osdp->sd_flags & FLG_SY_CAP) != 0)
3036         continue;
3037
3038     /*
3039      * As we're converting object capabilities to symbol
3040      * capabilities, obtain the capabilities set for this
3041      * object, so as to retrieve the CA_SUNW_ID value.
3042      */
3043     capset = &cppl->c_group->cg_set;
3044
3045     /*
3046      * Create a new name from the existing symbol and the
3047      * capabilities group identifier.  Note, the delimiter
3048      * between the symbol name and identifier name is hard-
3049      * coded here (%), so that we establish a convention
3050      * for transformed symbol names.
3051      */
3052     oname = osdp->sd_name;
3053
3054     idstr = capset->oc_id.cs_str;
3055     nsize = strlen(oname);
3056     tsize = nsize + 1 + strlen(idstr) + 1;
3057     if ((cname = libld_malloc(tsize)) == 0)

```

```

3058         return (S_ERROR);
3060         (void) strcpy(cname, oname);
3061         cname[nsize++] = '%';
3062         (void) strcpy(&cname[nsize], idstr);
3064         /*
3065          * Allocate a new symbol table entry, transform this
3066          * symbol to a local, and assign the new name.
3067          */
3068         if ((csym = libld_malloc(sizeof (Sym))) == NULL)
3069             return (S_ERROR);
3071         *csym = *osdp->sd_sym;
3072         csym->st_info = ELF_ST_INFO(STB_LOCAL,
3073             ELF_ST_TYPE(osdp->sd_sym->st_info));
3075         osdp->sd_name = cname;
3076         osdp->sd_sym = csym;
3077         osdp->sd_flags = FLG_SY_CAP;
3079         /*
3080          * Keep track of this new local symbol. As -z symbolcap
3081          * can only be used to create a relocatable object, a
3082          * dynamic symbol table can't exist. Ensure there is
3083          * space reserved in the string table.
3084          */
3085         ofl->ofl_caplocclnt++;
3086         if (st_insert(ofl->ofl_strtab, cname) == -1)
3087             return (S_ERROR);
3089         DBG_CALL(DBG_syms_cap_local(ofl, cppl->c_ndx,
3090             cname, csym, osdp));
3092         /*
3093          * Establish this capability pair as a family.
3094          */
3095         if (ld_cap_add_family(ofl, cppl->c_nsdp, osdp,
3096             cppl->c_group, &ifl->ifl_caps->ca_syms) == S_ERROR)
3097             return (S_ERROR);
3098     }
3099 }
3101     return (1);
3103 #undef SYM_LOC_BADADDR
3104 }
3106 /*
3107  * Add an undefined symbol to the symbol table. The reference originates from
3108  * the location identified by the message id (mid). These references can
3109  * originate from command line options such as -e, -u, -initarray, etc.
3110  * (identified with MSG_INTL(MSG_STR_COMMAND)), or from internally generated
3111  * TLS relocation references (identified with MSG_INTL(MSG_STR_TLSREL)).
3112  */
3113 Sym_desc *
3114 ld_sym_add_u(const char *name, Of1_desc *ofl, Msg mid)
3115 {
3116     Sym          *sym;
3117     Ifl_desc     *ifl = NULL, *_ifl;
3118     Sym_desc     *sdp;
3119     Word         hash;
3120     Aliste       idx;
3121     avl_index_t  where;
3122     const char   *reference = MSG_INTL(mid);

```

```

3124     /*
3125      * As an optimization, determine whether we've already generated this
3126      * reference. If the symbol doesn't already exist we'll create it.
3127      * Or if the symbol does exist from a different source, we'll resolve
3128      * the conflict.
3129      */
3130     /* LINTED */
3131     hash = (Word)elf_hash(name);
3132     if ((sdp = ld_sym_find(name, hash, &where, ofl)) != NULL) {
3133         if ((sdp->sd_sym->st_shndx == SHN_UNDEF) &&
3134             (sdp->sd_file->ifl_name == reference))
3135             return (sdp);
3136     }
3138     /*
3139      * Determine whether a pseudo input file descriptor exists to represent
3140      * the command line, as any global symbol needs an input file descriptor
3141      * during any symbol resolution (refer to map_ifl() which provides a
3142      * similar method for adding symbols from mapfiles).
3143      */
3144     for (APLIST_TRAVERSE(ofl->ofl_objs, idx, _ifl))
3145         if (strcmp(_ifl->ifl_name, reference) == 0) {
3146             ifl = _ifl;
3147             break;
3148         }
3150     /*
3151      * If no descriptor exists create one.
3152      */
3153     if (ifl == NULL) {
3154         if ((ifl = libld_calloc(sizeof (Ifl_desc), 1)) == NULL)
3155             return ((Sym_desc *)S_ERROR);
3156         ifl->ifl_name = reference;
3157         ifl->ifl_flags = FLG_IF_NEEDED | FLG_IF_FILEREF;
3158         if ((ifl->ifl_ehdr = libld_calloc(sizeof (Ehdr), 1)) == NULL)
3159             return ((Sym_desc *)S_ERROR);
3160         ifl->ifl_ehdr->e_type = ET_REL;
3162         if (aplist_append(&ofl->ofl_objs, ifl, AL_CNT_OF1_OBJS) == NULL)
3163             return ((Sym_desc *)S_ERROR);
3164     }
3166     /*
3167      * Allocate a symbol structure and add it to the global symbol table.
3168      */
3169     if ((sym = libld_calloc(sizeof (Sym), 1)) == NULL)
3170         return ((Sym_desc *)S_ERROR);
3171     sym->st_info = ELF_ST_INFO(STB_GLOBAL, STT_NOTYPE);
3172     sym->st_shndx = SHN_UNDEF;
3174     DBG_CALL(DBG_syms_process(ofl->ofl_lml, ifl));
3175     if (sdp == NULL) {
3176         DBG_CALL(DBG_syms_global(ofl->ofl_lml, 0, name));
3177         if ((sdp = ld_sym_enter(name, sym, hash, ifl, ofl, 0, SHN_UNDEF,
3178             0, &where)) == (Sym_desc *)S_ERROR)
3179             return ((Sym_desc *)S_ERROR);
3180     } else if (ld_sym_resolve(sdp, sym, ifl, ofl, 0,
3181         SHN_UNDEF, 0) == S_ERROR)
3182         return ((Sym_desc *)S_ERROR);
3184     sdp->sd_flags &= ~FLG_SY_CLEAN;
3185     sdp->sd_flags |= FLG_SY_CMDREF;
3187     return (sdp);
3188 }

```

```
3190 /*
3191  * STT_SECTION symbols have their st_name field set to NULL, and consequently
3192  * have no name. Generate a name suitable for diagnostic use for such a symbol
3193  * and store it in the input section descriptor. The resulting name will be
3194  * of the form:
3195  *
3196  *     "XXX (section)"
3197  *
3198  * where XXX is the name of the section.
3199  *
3200  * entry:
3201  *     isc - Input section associated with the symbol.
3202  *     fmt - NULL, or format string to use.
3203  *
3204  * exit:
3205  *     Sets isp->is_sym_name to the allocated string. Returns the
3206  *     string pointer, or NULL on allocation failure.
3207  */
3208 const char *
3209 ld_stt_section_sym_name(Is_desc *isp)
3210 {
3211     const char    *fmt;
3212     char          *str;
3213     size_t        len;
3214
3215     if ((isp == NULL) || (isp->is_name == NULL))
3216         return (NULL);
3217
3218     if (isp->is_sym_name == NULL) {
3219         fmt = (isp->is_flags & FLG_IS_GNSTRMRG) ?
3220             MSG_INTL(MSG_STR_SECTION_MSTR) : MSG_INTL(MSG_STR_SECTION);
3221
3222         len = strlen(fmt) + strlen(isp->is_name) + 1;
3223
3224         if ((str = libld_malloc(len)) == NULL)
3225             return (NULL);
3226         (void) snprintf(str, len, fmt, isp->is_name);
3227         isp->is_sym_name = str;
3228     }
3229
3230     return (isp->is_sym_name);
3231 }
```