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

PROG= file
XPG4PROG= file
MAGIC= magic

ELFCAP= $(SRC)/common/elfcap
SGSRTCID=       $(SRC)/common/sgsrtcid

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

include ../Makefile.cmd

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

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

# not linted
SMATCH=off

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

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

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

```
ROOTETCMAGIC= $(MAGIC:%=$(ROOTETC)/%)

$(ROOTETCMAGIC) :=      FILEMODE =      $(LIBFILEMODE)

.PARALLEL:      $(OBJS) $(XPG4OBJS) $(POFILES)

.KEEP_STATE:

all: $(PROG) $(XPG4) $(MAGIC)

$(PROG) : $(OBJS)
        $(LINK.c) $(OBJS) -o $@ $(LDLIBS)
        $(POST_PROCESS)

$(XPG4) : $(XPG4OBJS)
        $(LINK.c) $(XPG4OBJS) -o $@ $(LDLIBS)
        $(POST_PROCESS)

%.o:    %.c
        $(COMPILE.c) -o $@ $<

%32.o:  %.c
        $(COMPILE.c) -o $@ $<

%64.o:  %.c
        $(COMPILE.c) -D_ELF64 -o $@ $<

xpg4_%.o:       %.c
        $(COMPILE.c) -o $@ $<

xpg4_%32.o:     %.c
        $(COMPILE.c) -o $@ $<

xpg4_%64.o:     %.c
        $(COMPILE.c) -D_ELF64 -o $@ $<

elfcap.o:       $(ELFCAP)/elfcap.c
        $(COMPILE.c) -o $@ $(ELFCAP)/elfcap.c

xpg4_elfcap.o:  $(ELFCAP)/elfcap.c
        $(COMPILE.c) -o $@ $(ELFCAP)/elfcap.c

$(POFILE):      $(POFILES)
        $(RM) $@
        cat $(POFILES) > $@

install: all $(ROOTPROG) $(ROOTXPG4PROG) $(ROOTETCMAGIC)

clean:
        $(RM) $(OBJS) $(XPG4OBJS)

lint: lint_SRCS

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_*

```
/*
 * process_shdr:        Read Section Headers to attempt to get HW/SW
 *                      capabilities by looking at the SUNW_cap
 *                      section and set string in Elf_Info.
 *                      Also look for symbol tables and debug
 *                      information sections. Set the "stripped" field
 *                      in Elf_Info with corresponding flags.
 */
static int
process_shdr(Elf_Info *EI)
{
        int             mac;
        int             i, j, idx;
        char            *strtab;
        size_t          strtab_sz;
        Elf_Shdr        *shdr = &EI_Shdr;

        mac = EI_Ehdr.e_machine;

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

        /* read section names from String Section */
        if (get_shdr(EI, EI_Ehdr_shstrndx) == ELF_READ_FAIL)
                return (ELF_READ_FAIL);

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

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

        strtab_sz = shdr->sh_size;

        /* read all the sections and process them */
        for (idx = 1, i = 0; i < EI_Ehdr_shnum; idx++, i++) {
                char *shnam;

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

                if (shdr->sh_type == SHT_NULL) {
                        idx--;
                        continue;
                }

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

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

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

```
                                (void) fprintf(stderr, ELF_ERR_ELFCAP1,
                                    File, EI->file);
                                return (ELF_READ_FAIL);
                        }
                        capn = (shdr->sh_size / shdr->sh_entsize);
                        for (j = 0; j < capn; j++) {
                                /*
                                 * read cap and xlate the values
                                 */
                                if ((pread64(EI->elffd, &Chdr, csize, cap_off)
                                    != csize) ||
                                    file_xlatetom(ELF_T_CAP, (char *)&Chdr)
                                    == 0) {
                                        (void) fprintf(stderr, ELF_ERR_ELFCAP2,
                                            File, EI->file);
                                        return (ELF_READ_FAIL);
                                }

                                cap_off += csize;

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

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

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

                                (void) strlcat(EI->cap_str, capstr,
                                    sizeof (EI->cap_str));
                        }
                } else if (shdr->sh_type == SHT_DYNAMIC) {
                        Elf_Dyn dyn;
                        FILE_ELF_SIZE_T dsize;
                        FILE_ELF_OFF_T doff;
                        uint64_t dynn;
                        int dynn;

                        doff = shdr->sh_offset;
                        dsize = sizeof (Elf_Dyn);

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

                        dynn = (shdr->sh_size / shdr->sh_entsize);
                        for (j = 0; j < dynn; j++) {
                                if (pread64(EI->elffd, &dyn, dsize, doff)
                                    != dsize ||
                                    file_xlatetom(ELF_T_DYN, (char *)&dyn)
                                    == 0) {
                                        (void) fprintf(stderr, ELF_ERR_DYNAMIC2,
                                            File, EI->file);
                                        return (ELF_READ_FAIL);
                                }

                                doff += dsize;

                                if ((dyn.d_tag == DT_SUNW_KMOD) &&
                                    (dyn.d_un.d_val == 1)) {
                                        EI->kmod = B_TRUE;
                                }
                        }
                }

                /*
                 * Definition time:
                 *      - "not stripped" means that an executable file
                 *      contains a Symbol Table (.symtab)
                 *      - "stripped" means that an executable file
                 *      does not contain a Symbol Table.
                 * When strip -l or strip -x is run, it strips the
                 * debugging information (.line section name (strip -l),
                 * .line, .debug*, .stabs*, .dwarf* section names
                 * and SHT_SUNW_DEBUGSTR and SHT_SUNW_DEBUG
                 * section types (strip -x), however the Symbol
                 * Table will still be present.
                 * Therefore, if
                 *      - No Symbol Table present, then report
                 *              "stripped"
                 *      - Symbol Table present with debugging
                 *      information (line number or debug section names,
                 *      or SHT_SUNW_DEBUGSTR or SHT_SUNW_DEBUG section
                 *      types) then report:
                 *              "not stripped"
                 *      - Symbol Table present with no debugging
                 *      information (line number or debug section names,
                 *      or SHT_SUNW_DEBUGSTR or SHT_SUNW_DEBUG section
                 *      types) then report:
                 *              "not stripped, no debugging information
                 *              available"
                 */
                if ((EI->stripped & E_NOSTRIP) == E_NOSTRIP)
                        continue;

                if (!(EI->stripped & E_SYMTAB) &&
                    (shdr->sh_type == SHT_SYMTAB)) {
                        EI->stripped |= E_SYMTAB;
                        continue;
                }

                if (shdr->sh_name >= strtab_sz)
                        shnam = NULL;
                else
                        shnam = &strtab[shdr->sh_name];

                if (!(EI->stripped & E_DBGINF) &&
                    ((shdr->sh_type == SHT_SUNW_DEBUG) ||
                    (shdr->sh_type == SHT_SUNW_DEBUGSTR) ||
                    (shnam != NULL && is_in_list(shnam)))) {
                        EI->stripped |= E_DBGINF;
                }
        }
        free(strtab);

        return (ELF_READ_OKAY);
}
```
______***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_
```

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

static void
def_context_tests(void)
{
        int     j;
        int     nl;
        char    ch;
        int     len;

        if (ccom() == 0)
                goto notc;
        while (fbuf[i] == '#') {
                j = i;
                while (fbuf[i++] != '\n') {
                        if (i - j > 255) {
                                (void) printf(gettext("data\n"));
                                return;
                        }
                        if (i >= fbsz)
                                goto notc;
                }
                if (ccom() == 0)
                        goto notc;
        }
check:
        if (lookup(c) == 1) {
                while ((ch = fbuf[i]) != ';' && ch != '{') {
                        if ((len = mblen(&fbuf[i], MB_CUR_MAX)) <= 0)
                                len = 1;
                        i += len;
                        if (i >= fbsz)
                                goto notc;
                }
                (void) printf(gettext("c program text"));
                goto outa;
        }
        nl = 0;
        while (fbuf[i] != '(') {
                if (fbuf[i] <= 0)
                        goto notas;
                if (fbuf[i] == ';') {
                        i++;
                        goto check;
                }
                if (fbuf[i++] == '\n')
                        if (nl++ > 6)
                                goto notc;
                if (i >= fbsz)
                        goto notc;
        }
        while (fbuf[i] != ')') {
                if (fbuf[i++] == '\n')
                        if (nl++ > 6)
                                goto notc;
```

```
                if (i >= fbsz)
                        goto notc;
        }
        while (fbuf[i] != '{') {
                if ((len = mblen(&fbuf[i], MB_CUR_MAX)) <= 0)
                        len = 1;
                if (fbuf[i] == '\n')
                        if (nl++ > 6)
                                goto notc;
                i += len;
                if (i >= fbsz)
                        goto notc;
        }
        (void) printf(gettext("c program text"));
        goto outa;
notc:
        i = 0;                  /* reset to begining of file again */
        while (fbuf[i] == 'c' || fbuf[i] == 'C'|| fbuf[i] == '!' ||
            fbuf[i] == '*' || fbuf[i] == '\n') {
                while (fbuf[i++] != '\n')
                        if (i >= fbsz)
                                goto notfort;
        }
        if (lookup(fort) == 1) {
                (void) printf(gettext("fortran program text"));
                goto outa;
        }
notfort:                        /* looking for assembler program */
        i = 0;                  /* reset to beginning of file again */
        if (ccom() == 0)        /* assembler programs may contain */
                                /* c-style comments */
                goto notas;
        if (ascom() == 0)
                goto notas;
        j = i - 1;
        if (fbuf[i] == '.') {
                i++;
                if (lookup(as) == 1) {
                        (void) printf(gettext("assembler program text"));
                        goto outa;
                } else if (j != -1 && fbuf[j] == '\n' && isalpha(fbuf[j + 2])) {
                        (void) printf(
                            gettext("[nt]roff, tbl, or eqn input text"));
                        goto outa;
                }
        }
        while (lookup(asc) == 0) {
                if (ccom() == 0)
                        goto notas;
                if (ascom() == 0)
                        goto notas;
                while (fbuf[i] != '\n' && fbuf[i++] != ':') {
                        if (i >= fbsz)
                                goto notas;
                }
                while (fbuf[i] == '\n' || fbuf[i] == ' ' || fbuf[i] == '\t')
                        if (i++ >= fbsz)
                                goto notas;
                j = i - 1;
                if (fbuf[i] == '.') {
                        i++;
                        if (lookup(as) == 1) {
                                (void) printf(
                                    gettext("assembler program text"));
                                goto outa;
                        } else if (fbuf[j] == '\n' && isalpha(fbuf[j+2])) {
                                (void) printf(
                                    gettext("[nt]roff, tbl, or eqn input "
                                    "text"));
                                goto outa;
                        }
                }
        }
        (void) printf(gettext("assembler program text"));
        goto outa;
notas:
        /* start modification for multibyte env */
        IS_ascii = 1;
        if (fbsz < FBSZ)
                Max = fbsz;
        else
                Max = FBSZ - MB_LEN_MAX; /* prevent cut of wchar read */
        /* end modification for multibyte env */

        for (i = 0; i < Max; /* null */)
                if (fbuf[i] & 0200) {
                        IS_ascii = 0;
                        if ((fbuf[0] == '\100') &&
                            ((uchar_t)fbuf[1] == (uchar_t)'\357')) {
                        if (fbuf[0] == '\100' && fbuf[1] == '\357') {
                                (void) printf(gettext("troff output\n"));
                                return;
                        }
                /* start modification for multibyte env */
                        if ((length = mbtowc(&wchar, &fbuf[i], MB_CUR_MAX))
                            <= 0 || !iswprint(wchar)) {
                                (void) printf(gettext("data\n"));
                                return;
                        }
                        i += length;
                }
                else
                        i++;
        i = fbsz;
                /* end modification for multibyte env */
        if (mbuf.st_mode&(S_IXUSR|S_IXGRP|S_IXOTH))
                (void) printf(gettext("commands text"));
        else if (troffint(fbuf, fbsz))
                (void) printf(gettext("troff intermediate output text"));
        else if (english(fbuf, fbsz))
                (void) printf(gettext("English text"));
        else if (IS_ascii)
                (void) printf(gettext("ascii text"));
        else
                (void) printf(gettext("text")); /* for multibyte env */
outa:
        /*
         * This code is to make sure that no MB char is cut in half
         * while still being used.
         */
        fbsz = (fbsz < FBSZ ? fbsz : fbsz - MB_CUR_MAX + 1);
        while (i < fbsz) {
                if (isascii(fbuf[i])) {
                        i++;
                        continue;
                } else {
                        if ((length = mbtowc(&wchar, &fbuf[i], MB_CUR_MAX))
                            <= 0 || !iswprint(wchar)) {
                                (void) printf(gettext(" with garbage\n"));
                                return;
                        }
                        i = i + length;
```

```
                }
        }
        (void) printf("\n");
}
```

______***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
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

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

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

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

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <limits.h>
#include <inttypes.h>
#include <sys/types.h>
#include <libintl.h>

/*
 *      Types
 */

#define BYTE    1
#define SHORT   2
#define LONG    4
#define LLONG   8
#define UBYTE   16
#define USHORT  32
#define ULONG   64
#define ULLONG  128
#define STR     256

/*
 *      Opcodes
 */
```

```
#define EQ      0
#define GT      1
#define LT      2
#define STRC    3       /* string compare */
#define ANY     4
#define AND     5
#define NSET    6       /* True if bit is not set */
#define SUB     64      /* or'ed in, SUBstitution string, for example */
                        /* %ld, %s, %lo mask: with bit 6 on, used to locate */
                        /* print formats */
/*
 *      Misc
 */

#define BSZ     128
#define NENT    200

/*
 *      Structure of magic file entry
 */

struct  entry   {
        char            e_level;        /* 0 or 1 */
        off_t           e_off;          /* in bytes */
        uint32_t        e_type;         /* BYTE, SHORT, STR, et al */
        char            e_opcode;       /* EQ, GT, LT, ANY, AND, NSET */
        uint64_t        e_mask;         /* if non-zero, mask value with this */
        union   {
                uint64_t        num;
                char            *str;
        }       e_value;
        const char      *e_str;
};
______unchanged_portion_omitted_

/*
 * f_mkmtab - fills mtab array of magic table entries with
 *      values from the file magfile.
 *      May be called more than once if multiple magic
 *      files were specified.
 *      Stores entries sequentially in one of two magic
 *      tables: mtab1, if first = 1; mtab2 otherwise.
 *
 *      If -c option is specified, cflg is non-zero, and
 *      f_mkmtab() reports on errors in the magic file.
 *
 *      Two magic tables may need to be created.  The first
 *      one (mtab1) contains magic entries to be checked before
 *      the programmatic default position-sensitive tests in
 *      def_position_tests().
 *      The second one (mtab2) should start with the default
 *      /etc/magic file entries and is to be checked after
 *      the programmatic default position-sensitive tests in
 *      def_position_tests().  The parameter "first" would
 *      be 1 for the former set of tables, 0 for the latter
 *      set of magic tables.
 *      No mtab2 should be created if file will not be
 *      applying default tests; in that case, all magic table
 *      entries should be in mtab1.
 *
 *      f_mkmtab returns 0 on success, -1 on error.  The calling
 *      program is not expected to proceed after f_mkmtab()
 *      returns an error.
 */

int
```

```
f_mkmtab(char *magfile, int cflg, int first)
{
        Entry   *mtab;  /* generic magic table pointer */
        Entry   *ep;    /* current magic table entry */
        Entry   *mend;  /* one past last-allocated entry of mtab */
        FILE    *fp;
        int     lcnt = 0;
        char    buf[BSZ];
        size_t  tbsize;
        size_t  oldsize;

        if (first) {
                mtab = mtab1;
                mend = mend1;
                ep = ep1;
        } else {
                mtab = mtab2;
                mend = mend2;
                ep = ep2;
        }

        /* mtab may have been allocated on a previous f_mkmtab call */
        if (mtab == (Entry *)NULL) {
                if ((mtab = calloc(NENT, sizeof (Entry))) == NULL) {
                if ((mtab = calloc(sizeof (Entry), NENT)) == NULL) {
                        int err = errno;
                        (void) fprintf(stderr, gettext("%s: malloc "
                            "failed: %s\n"), File, strerror(err));
                        return (-1);
                }

                ep = mtab;
                mend = &mtab[NENT];
        }

        errno = 0;
        if ((fp = fopen(magfile, "r")) == NULL) {
                int err = errno;
                (void) fprintf(stderr, gettext("%s: %s: cannot open magic "
                    "file: %s\n"), File, magfile, err ? strerror(err) : "");
                return (-1);
        }
        while (fgets(buf, BSZ, fp) != NULL) {
                char    *p = buf;
                char    *p2;
                char    *p3;
                char    opc;

                /*
                 * ensure we have one extra entry allocated
                 * to mark end of the table, after the while loop
                 */
                if (ep >= (mend - 1)) {
                        oldsize = mend - mtab;
                        tbsize = (NENT + oldsize) * sizeof (Entry);
                        if ((mtab = realloc(mtab, tbsize)) == NULL) {
                                int err = errno;
                                (void) fprintf(stderr, gettext("%s: malloc "
                                    "failed: %s\n"), File, strerror(err));
                                return (-1);
                        } else {
                                (void) memset(mtab + oldsize, 0,
                                    sizeof (Entry) * NENT);
                                mend = &mtab[tbsize / sizeof (Entry)];
                                ep = &mtab[oldsize-1];
                        }
```

```
                }

                lcnt++;
                if (*p == ’\n’ || *p == ’#’)
                        continue;


                        /* LEVEL */
                if (*p == ’>’) {
                        ep->e_level = 1;
                        p++;
                }
                        /* OFFSET */
                p2 = strchr(p, ’\t’);
                if (p2 == NULL) {
                        if (cflg)
                                (void) fprintf(stderr, gettext("%s: %s: format "
                                    "error, no tab after %s on line %d\n"),
                                    File, magfile, p, lcnt);
                        continue;
                }
                *p2++ = NULL;
                ep->e_off = strtol((const char *)p, (char **)NULL, 0);
                while (*p2 == ’\t’)
                        p2++;
                        /* TYPE */
                p = p2;
                p2 = strchr(p, ’\t’);
                if (p2 == NULL) {
                        if (cflg)
                                (void) fprintf(stderr, gettext("%s: %s: format "
                                    "error, no tab after %s on line %d\n"),
                                    File, magfile, p, lcnt);
                        continue;
                }
                *p2++ = NULL;
                p3 = strchr(p, ’&’);
                if (p3 != NULL) {
                        *p3++ = ’\0’;
                        ep->e_mask = strtoull((const char *)p3, (char **)NULL,
                            0); /* returns 0 or ULLONG_MAX on error */
                } else {
                        ep->e_mask = 0ULL;
                }
                switch (*p) {
                        case ’d’:
                                if (*(p+1) == NULL) {
                                        /* d */
                                        ep->e_type = LONG;
                                } else if (*(p+2) == NULL) {    /* d? */
                                        switch (*(p+1)) {
                                                case ’C’:
                                                case ’1’:
                                                        /* dC, d1 */
                                                        ep->e_type = BYTE;
                                                        break;
                                                case ’S’:
                                                case ’2’:
                                                        /* dS, d2 */
                                                        ep->e_type = SHORT;
                                                        break;
                                                case ’I’:
                                                case ’L’:
                                                case ’4’:
                                                        /* dI, dL, d4 */
                                                        ep->e_type = LONG;
                                                        break;
                                                case ’8’:
                                                        /* d8 */
                                                        ep->e_type = LLONG;
                                                        break;
                                                default:
                                                        ep->e_type = LONG;
                                                        break;
                                        }
                                }
                                break;
                        case ’l’:
                                if (*(p+1) == ’l’) {    /* llong */
                                        ep->e_type = LLONG;
                                } else {                /* long */
                                        ep->e_type = LONG;
                                }
                                break;
                        case ’s’:
                                if (*(p+1) == ’h’) {
                                        /* short */
                                        ep->e_type = SHORT;
                                } else {
                                        /* s or string */
                                        ep->e_type = STR;
                                }
                                break;
                        case ’u’:
                                if (*(p+1) == NULL) {
                                        /* u */
                                        ep->e_type = ULONG;
                                } else if (*(p+2) == NULL) {    /* u? */
                                        switch (*(p+1)) {
                                                case ’C’:
                                                case ’1’:
                                                        /* uC, u1 */
                                                        ep->e_type = UBYTE;
                                                        break;
                                                case ’S’:
                                                case ’2’:
                                                        /* uS, u2 */
                                                        ep->e_type = USHORT;
                                                        break;
                                                case ’I’:
                                                case ’L’:
                                                case ’4’:
                                                        /* uI, uL, u4 */
                                                        ep->e_type = ULONG;
                                                        break;
                                                case ’8’:
                                                        /* u8 */
                                                        ep->e_type = ULLONG;
                                                        break;
                                                default:
                                                        ep->e_type = ULONG;
                                                        break;
                                        }
                                } else { /* u?* */
                                        switch (*(p+1)) {
                                        case ’b’:       /* ubyte */
                                                ep->e_type = UBYTE;
                                                break;
                                        case ’s’:       /* ushort */
                                                ep->e_type = USHORT;
                                                break;
                                        case ’l’:
```

```
                                                if (*(p+2) == ’l’) {
                                                        /* ullong */
                                                        ep->e_type = ULLONG;
                                                } else {
                                                        /* ulong */
                                                        ep->e_type = ULONG;
                                                }
                                                break;
                                        default:
                                                /* default, same as "u" */
                                                ep->e_type = ULONG;
                                                break;
                                        }
                                }
                                break;
                        default:
                                /* retain (undocumented) default type */
                                ep->e_type = BYTE;
                                break;
                }
                if (ep->e_type == 0) {
                        ep->e_type = BYTE;      /* default */
                }
                while (*p2 == ’\t’)
                        p2++;
                        /* OP-VALUE */
                p = p2;
                p2 = strchr(p, ’\t’);
                if (p2 == NULL) {
                        if (cflg)
                                (void) fprintf(stderr, gettext("%s: %s: format "
                                    "error, no tab after %s on line %d\n"),
                                    File, magfile, p, lcnt);
                        continue;
                }
                *p2++ = NULL;
                if (ep->e_type != STR) {
                        opc = *p++;
                        switch (opc) {
                        case ’=’:
                                ep->e_opcode = EQ;
                                break;

                        case ’>’:
                                ep->e_opcode = GT;
                                break;

                        case ’<’:
                                ep->e_opcode = LT;
                                break;

                        case ’x’:
                                ep->e_opcode = ANY;
                                break;

                        case ’&’:
                                ep->e_opcode = AND;
                                break;

                        case ’^’:
                                ep->e_opcode = NSET;
                                break;
                        default:        /* EQ (i.e. 0) is default       */
                                p--;    /* since global ep->e_opcode=0  */
                        }
                }
```

```
		if (ep->e_opcode != ANY) {
			if (ep->e_type != STR) {
				ep->e_value.num = strtoull((const char *)p,
				    (char **)NULL, 0);
			} else if ((ep->e_value.str =
			    getstr(p, magfile)) == NULL) {
				return (-1);
			}
		}
		p2 += strspn(p2, "\t");
			/* STRING */
		if ((ep->e_str = strdup(p2)) == NULL) {
			int err = errno;
			(void) fprintf(stderr, gettext("%s: malloc "
			    "failed: %s\n"), File, strerror(err));
			return (-1);
		} else {
			if ((p = strchr(ep->e_str, '\n')) != NULL)
				*p = '\0';
			if (strchr(ep->e_str, '%') != NULL)
				ep->e_opcode |= SUB;
		}
		ep++;
	}	/* end while (fgets) */

	ep->e_off = -1L;	/* mark end of table */
	if (first) {
		mtab1 = mtab;
		mend1 = mend;
		ep1 = ep;
	} else {
		mtab2 = mtab;
		mend2 = mend;
		ep2 = ep;
	}
	if (fclose(fp) != 0) {
		int err = errno;
		(void) fprintf(stderr, gettext("%s: fclose failed: %s\n"),
		    File, strerror(err));
		return (-1);
	}
	return (0);
}
```
______***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_
```

```
static int	optitle = 0;
/*
 * Parsing options pass1 for process_flags().
 */
static uintptr_t
parseopt_pass1(Ofl_desc *ofl, int argc, char **argv, int *usage)
{
	int	c, ndx = optind;

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

		switch (c) {
		case '3':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));

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

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

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

		case 'a':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
			aflag = TRUE;
			break;

		case 'b':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
			bflag = TRUE;
```

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

		case 'c':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
			if (ofl->ofl_config)
				ld_eprintf(ofl, ERR_WARNING_NF,
				    MSG_INTL(MSG_ARG_MTONCE),
				    MSG_ORIG(MSG_ARG_C));
			else
				ofl->ofl_config = optarg;
			break;

		case 'C':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
			demangle_flag = 1;
			break;

		case 'd':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
			if ((optarg[0] == 'n') && (optarg[1] == '\0')) {
				if (dflag != SET_UNKNOWN)
					ld_eprintf(ofl, ERR_WARNING_NF,
					    MSG_INTL(MSG_ARG_MTONCE),
					    MSG_ORIG(MSG_ARG_D));
				else
					dflag = SET_FALSE;
			} else if ((optarg[0] == 'y') && (optarg[1] == '\0')) {
				if (dflag != SET_UNKNOWN)
					ld_eprintf(ofl, ERR_WARNING_NF,
					    MSG_INTL(MSG_ARG_MTONCE),
					    MSG_ORIG(MSG_ARG_D));
				else
					dflag = SET_TRUE;
			} else {
				ld_eprintf(ofl, ERR_FATAL,
				    MSG_INTL(MSG_ARG_ILLEGAL),
				    MSG_ORIG(MSG_ARG_D), optarg);
			}
			break;

		case 'e':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
			if (ofl->ofl_entry)
				ld_eprintf(ofl, ERR_WARNING_NF,
				    MSG_INTL(MSG_MARG_MTONCE),
				    MSG_INTL(MSG_MARG_ENTRY));
			else
				ofl->ofl_entry = (void *)optarg;
			break;

		case 'f':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
			if (ofl->ofl_filtees &&
			    (!(ofl->ofl_flags & FLG_OF_AUX))) {
				ld_eprintf(ofl, ERR_FATAL,
					    MSG_INTL(MSG_MARG_INCOMP),
					    MSG_INTL(MSG_MARG_FILTER_AUX),
					    MSG_INTL(MSG_MARG_FILTER));
			} else {
				if ((ofl->ofl_filtees =
				    add_string(ofl->ofl_filtees, optarg)) ==
				    (const char *)S_ERROR)
					return (S_ERROR);
				ofl->ofl_flags |= FLG_OF_AUX;
			}
			break;

		case 'F':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
			if (ofl->ofl_filtees &&
			    (ofl->ofl_flags & FLG_OF_AUX)) {
				ld_eprintf(ofl, ERR_FATAL,
				    MSG_INTL(MSG_MARG_INCOMP),
				    MSG_INTL(MSG_MARG_FILTER),
				    MSG_INTL(MSG_MARG_FILTER_AUX));
			} else {
				if ((ofl->ofl_filtees =
				    add_string(ofl->ofl_filtees, optarg)) ==
				    (const char *)S_ERROR)
					return (S_ERROR);
			}
			break;

		case 'h':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
			if (ofl->ofl_soname)
				ld_eprintf(ofl, ERR_WARNING_NF,
				    MSG_INTL(MSG_MARG_MTONCE),
				    MSG_INTL(MSG_MARG_SONAME));
			else
				ofl->ofl_soname = (const char *)optarg;
			break;

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

		case 'I':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
			if (ofl->ofl_interp)
				ld_eprintf(ofl, ERR_WARNING_NF,
				    MSG_INTL(MSG_ARG_MTONCE),
				    MSG_ORIG(MSG_ARG_CI));
			else
				ofl->ofl_interp = (const char *)optarg;
			break;

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

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

```
			ofl->ofl_flags |= FLG_OF_GENMAP;
			break;

		case 'o':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
			if (ofl->ofl_name)
				ld_eprintf(ofl, ERR_WARNING_NF,
				    MSG_INTL(MSG_MARG_MTONCE),
				    MSG_INTL(MSG_MARG_OUTFILE));
			else
				ofl->ofl_name = (const char *)optarg;
			break;

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

			/*
			 * Multiple instances of this option may occur.  Each
			 * additional instance is effectively concatenated to
			 * the previous separated by a colon.
			 */
			if (*optarg != '\0') {
				if ((ofl->ofl_audit =
				    add_string(ofl->ofl_audit,
				    optarg)) == (const char *)S_ERROR)
					return (S_ERROR);
			}
			break;

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

			/*
			 * Multiple instances of this option may occur.  Each
			 * additional instance is effectively concatenated to
			 * the previous separated by a colon.
			 */
			if (*optarg != '\0') {
				if ((ofl->ofl_depaudit =
				    add_string(ofl->ofl_depaudit,
				    optarg)) == (const char *)S_ERROR)
					return (S_ERROR);
			}
			break;

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

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

			/*
			 * Multiple instances of this option may occur.  Each
			 * additional instance is effectively concatenated to
			 * the previous separated by a colon.
			 */
			if (*optarg != '\0') {
				if ((ofl->ofl_rpath =
				    add_string(ofl->ofl_rpath,
				    optarg)) == (const char *)S_ERROR)
					return (S_ERROR);
			}
			break;
```

```
                case 's':
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
                        sflag = TRUE;
                        break;

                case 't':
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
                        ofl->ofl_flags |= FLG_OF_NOWARN;
                        break;

                case 'u':
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
                        break;

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

                        /*
                         * Skip comma that might be present between -z and its
                         * argument (e.g. if -Wl,-z,assert-deflib was passed).
                         */
                        if (strncmp(optarg, MSG_ORIG(MSG_STR_COMMA),
                            MSG_STR_COMMA_SIZE) == 0)
                                optarg++;

                        /*
                         * For specific help, print our usage message and exit
                         * immediately to ensure a 0 return code.
                         */
                        if (strncmp(optarg, MSG_ORIG(MSG_ARG_HELP),
                            MSG_ARG_HELP_SIZE) == 0) {
                                usage_mesg(TRUE);
                                exit(0);
                        }

                        /*
                         * For some options set a flag - further consistancy
                         * checks will be carried out in check_flags().
                         */
                        if ((strncmp(optarg, MSG_ORIG(MSG_ARG_LD32),
                            MSG_ARG_LD32_SIZE) == 0) ||
                            (strncmp(optarg, MSG_ORIG(MSG_ARG_LD64),
                            MSG_ARG_LD64_SIZE) == 0)) {
                                if (createargv(ofl, usage) == S_ERROR)
                                        return (S_ERROR);

                        } else if (
                            strcmp(optarg, MSG_ORIG(MSG_ARG_DEFS)) == 0) {
                                if (zdflag != SET_UNKNOWN)
                                        ld_eprintf(ofl, ERR_WARNING_NF,
                                            MSG_INTL(MSG_ARG_MTONCE),
                                            MSG_ORIG(MSG_ARG_ZDEFNODEF));
                                else
                                        zdflag = SET_TRUE;
                                ofl->ofl_guideflags |= FLG_OFG_NO_DEFS;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NODEFS)) == 0) {
                                if (zdflag != SET_UNKNOWN)
                                        ld_eprintf(ofl, ERR_WARNING_NF,
                                            MSG_INTL(MSG_ARG_MTONCE),
                                            MSG_ORIG(MSG_ARG_ZDEFNODEF));
                                else
                                        zdflag = SET_FALSE;
                                ofl->ofl_guideflags |= FLG_OFG_NO_DEFS;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_TEXT)) == 0) {
```

```
                                if (ztflag &&
                                    (ztflag != MSG_ORIG(MSG_ARG_ZTEXT)))
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_INCOMP),
                                            MSG_ORIG(MSG_ARG_ZTEXT),
                                            ztflag);
                                ztflag = MSG_ORIG(MSG_ARG_ZTEXT);
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_TEXTOFF)) == 0) {
                                if (ztflag &&
                                    (ztflag != MSG_ORIG(MSG_ARG_ZTEXTOFF)))
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_INCOMP),
                                            MSG_ORIG(MSG_ARG_ZTEXTOFF),
                                            ztflag);
                                ztflag = MSG_ORIG(MSG_ARG_ZTEXTOFF);
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_TEXTWARN)) == 0) {
                                if (ztflag &&
                                    (ztflag != MSG_ORIG(MSG_ARG_ZTEXTWARN)))
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_INCOMP),
                                            MSG_ORIG(MSG_ARG_ZTEXTWARN),
                                            ztflag);
                                ztflag = MSG_ORIG(MSG_ARG_ZTEXTWARN);

                        /*
                         * For other options simply set the ofl flags directly.
                         */
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_RESCAN)) == 0) {
                                ofl->ofl_flags1 |= FLG_OF1_RESCAN;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_ABSEXEC)) == 0) {
                                ofl->ofl_flags1 |= FLG_OF1_ABSEXEC;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_LOADFLTR)) == 0) {
                                zlflag = TRUE;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NORELOC)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_NORELOC;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOVERSION)) == 0) {
                                ofl->ofl_flags |= FLG_OF_NOVERSEC;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_MULDEFS)) == 0) {
                                ofl->ofl_flags |= FLG_OF_MULDEFS;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_REDLOCSYM)) == 0) {
                                ofl->ofl_flags |= FLG_OF_REDLSYM;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_INITFIRST)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_INITFIRST;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NODELETE)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_NODELETE;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOPARTIAL)) == 0) {
                                ofl->ofl_flags1 |= FLG_OF1_NOPARTI;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOOPEN)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_NOOPEN;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOW)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_NOW;
                                ofl->ofl_dtflags |= DF_BIND_NOW;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_ORIGIN)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
                                ofl->ofl_dtflags |= DF_ORIGIN;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NODEFAULTLIB)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_NODEFLIB;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NODUMP)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_NODUMP;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_ENDFILTEE)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_ENDFILTEE;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_VERBOSE)) == 0) {
                                ofl->ofl_flags |= FLG_OF_VERBOSE;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_COMBRELOC)) == 0) {
                                ofl->ofl_flags |= FLG_OF_COMREL;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOCOMBRELOC)) == 0) {
                                ofl->ofl_flags |= FLG_OF_NOCOMREL;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOCOMPSTRTAB)) == 0) {
                                ofl->ofl_flags1 |= FLG_OF1_NCSTTAB;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOINTERP)) == 0) {
                                ofl->ofl_flags1 |= FLG_OF1_NOINTRP;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_INTERPOSE)) == 0) {
                                zinflag = TRUE;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_IGNORE)) == 0) {
                                ofl->ofl_flags1 |= FLG_OF1_IGNPRC;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_RELAXRELOC)) == 0) {
                                ofl->ofl_flags1 |= FLG_OF1_RLXREL;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NORELAXRELOC)) == 0) {
                                ofl->ofl_flags1 |= FLG_OF1_NRLXREL;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOLDYNSYM)) == 0) {
                                ofl->ofl_flags |= FLG_OF_NOLDYNSYM;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_GLOBAUDIT)) == 0) {
                                ofl->ofl_dtflags_1 |= DF_1_GLOBAUDIT;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOSIGHANDLER)) == 0) {
                                ofl->ofl_flags1 |= FLG_OF1_NOSGHND;
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_SYMBOLCAP)) == 0) {
                                ofl->ofl_flags |= FLG_OF_OTOSCAP;

                        /*
                         * Check archive group usage
                         *      -z rescan-start ... -z rescan-end
                         * to ensure they don't overlap and are well formed.
                         */
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_RESCAN_START)) == 0) {
                                if (ofl->ofl_ars_gsandx == 0) {
                                        ofl->ofl_ars_gsandx = ndx;
                                } else if (ofl->ofl_ars_gsandx > 0) {
                                        /* Another group is still open */
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_AR_GRP_OLAP),
```

```
                                            MSG_INTL(MSG_MARG_AR_GRPS));
                                        /* Don't report cascading errors */
                                        ofl->ofl_ars_gsandx = -1;
                                }
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_RESCAN_END)) == 0) {
                                if (ofl->ofl_ars_gsandx > 0) {
                                        ofl->ofl_ars_gsandx = 0;
                                } else if (ofl->ofl_ars_gsandx == 0) {
                                        /* There was no matching begin */
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_AR_GRP_BAD),
                                            MSG_INTL(MSG_MARG_AR_GRP_END),
                                            MSG_INTL(MSG_MARG_AR_GRP_START));
                                        /* Don't report cascading errors */
                                        ofl->ofl_ars_gsandx = -1;
                                }

                        /*
                         * If -z wrap is seen, enter the symbol to be wrapped
                         * into the wrap AVL tree.
                         */
                        } else if (strncmp(optarg, MSG_ORIG(MSG_ARG_WRAP),
                            MSG_ARG_WRAP_SIZE) == 0) {
                                if (ld_wrap_enter(ofl,
                                    optarg + MSG_ARG_WRAP_SIZE) == NULL)
                                        return (S_ERROR);
                        } else if (strncmp(optarg, MSG_ORIG(MSG_ARG_ASLR),
                            MSG_ARG_ASLR_SIZE) == 0) {
                                char *p = optarg + MSG_ARG_ASLR_SIZE;
                                if (*p == '\0') {
                                        ofl->ofl_aslr = 1;
                                } else if (*p == '=') {
                                        p++;

                                        if ((strcmp(p,
                                            MSG_ORIG(MSG_ARG_ENABLED)) == 0) ||
                                            (strcmp(p,
                                            MSG_ORIG(MSG_ARG_ENABLE)) == 0)) {
                                                ofl->ofl_aslr = 1;
                                        } else if ((strcmp(p,
                                            MSG_ORIG(MSG_ARG_DISABLED)) == 0) ||
                                            (strcmp(p,
                                            MSG_ORIG(MSG_ARG_DISABLE)) == 0)) {
                                                ofl->ofl_aslr = -1;
                                        } else {
                                                ld_eprintf(ofl, ERR_FATAL,
                                                    MSG_INTL(MSG_ARG_ILLEGAL),
                                                    MSG_ORIG(MSG_ARG_ZASLR), p);
                                                return (S_ERROR);
                                        }
                                } else {
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_ILLEGAL),
                                            MSG_ORIG(MSG_ARG_Z), optarg);
                                        return (S_ERROR);
                                }
                        } else if ((strncmp(optarg, MSG_ORIG(MSG_ARG_GUIDE),
                            MSG_ARG_GUIDE_SIZE) == 0) &&
                            ((optarg[MSG_ARG_GUIDE_SIZE] == '=') ||
                            (optarg[MSG_ARG_GUIDE_SIZE] == '\0'))) {
                                if (!guidance_parse(ofl, optarg))
                                        return (S_ERROR);
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_FATWARN)) == 0) {
                                if (zfwflag  == SET_FALSE) {
                                        ld_eprintf(ofl, ERR_WARNING_NF,
                                            MSG_INTL(MSG_ARG_MTONCE),
                                            MSG_ORIG(MSG_ARG_ZFATWNOFATW));
                                } else {
                                        zfwflag = SET_TRUE;
                                        ofl->ofl_flags |= FLG_OF_FATWARN;
                                }
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NOFATWARN)) == 0) {
                                if (zfwflag  == SET_TRUE)
                                        ld_eprintf(ofl, ERR_WARNING_NF,
                                            MSG_INTL(MSG_ARG_MTONCE),
                                            MSG_ORIG(MSG_ARG_ZFATWNOFATW));
                                else
                                        zfwflag = SET_FALSE;

                        /*
                         * Process everything related to -z assert-deflib. This
                         * must be done in pass 1 because it gets used in pass
                         * 2.
                         */
                        } else if (strncmp(optarg, MSG_ORIG(MSG_ARG_ASSDEFLIB),
                            MSG_ARG_ASSDEFLIB_SIZE) == 0) {
                                if (assdeflib_parse(ofl, optarg) != TRUE)
                                        return (S_ERROR);

                        /*
                         * Process new-style output type specification, which
                         * we’ll use in pass 2 and throughout.
                         */
#endif /* ! codereview */
                        } else if (strncmp(optarg, MSG_ORIG(MSG_ARG_TYPE),
                            MSG_ARG_TYPE_SIZE) == 0) {
                                char *p = optarg + MSG_ARG_TYPE_SIZE;
                                if (*p != ’=’) {
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_ILLEGAL),
                                            MSG_ORIG(MSG_ARG_Z), optarg);
                                        return (S_ERROR);
                                }

                                p++;
                                if (strcmp(p,
                                    MSG_ORIG(MSG_ARG_TYPE_RELOC)) == 0) {
                                        otype = OT_RELOC;
                                } else if (strcmp(p,
                                    MSG_ORIG(MSG_ARG_TYPE_EXEC)) == 0) {
                                        otype = OT_EXEC;
                                } else if (strcmp(p,
                                    MSG_ORIG(MSG_ARG_TYPE_SHARED)) == 0) {
                                        otype = OT_SHARED;
                                } else if (strcmp(p,
                                    MSG_ORIG(MSG_ARG_TYPE_KMOD)) == 0) {
                                        otype = OT_KMOD;
                                } else {
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_ILLEGAL),
                                            MSG_ORIG(MSG_ARG_Z), optarg);
                                        return (S_ERROR);
                                }
                        /*
                         * The following options just need validation as they
                         * are interpreted on the second pass through the
                         * command line arguments.
                         */
                        } else if (
```

```
                            strncmp(optarg, MSG_ORIG(MSG_ARG_INITARRAY),
                            MSG_ARG_INITARRAY_SIZE) &&
                            strncmp(optarg, MSG_ORIG(MSG_ARG_FINIARRAY),
                            MSG_ARG_FINIARRAY_SIZE) &&
                            strncmp(optarg, MSG_ORIG(MSG_ARG_PREINITARRAY),
                            MSG_ARG_PREINITARRAY_SIZE) &&
                            strncmp(optarg, MSG_ORIG(MSG_ARG_RTLDINFO),
                            MSG_ARG_RTLDINFO_SIZE) &&
                            strncmp(optarg, MSG_ORIG(MSG_ARG_DTRACE),
                            MSG_ARG_DTRACE_SIZE) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_ALLEXTRT)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_DFLEXTRT)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_DIRECT)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_NODIRECT)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_GROUPPERM)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_LAZYLOAD)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_NOGROUPPERM)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_NOLAZYLOAD)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_NODEFERRED)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_RECORD)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_ALTEXEC64)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_WEAKEXT)) &&
                            strncmp(optarg, MSG_ORIG(MSG_ARG_TARGET),
                            MSG_ARG_TARGET_SIZE) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_RESCAN_NOW)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_DEFERRED))) {
                                ld_eprintf(ofl, ERR_FATAL,
                                    MSG_INTL(MSG_ARG_ILLEGAL),
                                    MSG_ORIG(MSG_ARG_Z), optarg);
                        }

                        break;

                case ’D’:
                        /*
                         * If we have not yet read any input files go ahead
                         * and process any debugging options (this allows any
                         * argument processing, entrance criteria and library
                         * initialization to be displayed).  Otherwise, if an
                         * input file has been seen, skip interpretation until
                         * process_files (this allows debugging to be turned
                         * on and off around individual groups of files).
                         */
                        Dflag = 1;
                        if (ofl->ofl_objscnt == 0) {
                                if (dbg_setup(ofl, optarg, 2) == 0)
                                        return (S_ERROR);
                        }

                        /*
                         * A diagnostic can only be provided after dbg_setup().
                         * As this is the first diagnostic that can be produced
                         * by ld(1), issue a title for timing and basic output.
                         */
                        if ((optitle == 0) && DBG_ENABLED) {
                                optitle++;
                                DBG_CALL(Dbg_basic_options(ofl->ofl_lml));
                        }
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
                        break;

                case ’B’:
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
                        if (strcmp(optarg, MSG_ORIG(MSG_ARG_DIRECT)) == 0) {
                                if (Bdflag == SET_FALSE) {
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_INCOMP),
                                            MSG_ORIG(MSG_ARG_BNODIRECT),
                                            MSG_ORIG(MSG_ARG_BDIRECT));
                                } else {
                                        Bdflag = SET_TRUE;
                                        ofl->ofl_guideflags |= FLG_OFG_NO_DB;
                                }
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_NODIRECT)) == 0) {
                                if (Bdflag == SET_TRUE) {
                                        ld_eprintf(ofl, ERR_FATAL,
                                            MSG_INTL(MSG_ARG_INCOMP),
                                            MSG_ORIG(MSG_ARG_BDIRECT),
                                            MSG_ORIG(MSG_ARG_BNODIRECT));
                                } else {
                                        Bdflag = SET_FALSE;
                                        ofl->ofl_guideflags |= FLG_OFG_NO_DB;
                                }
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_STR_SYMBOLIC)) == 0)
                                Bsflag = TRUE;
                        else if (strcmp(optarg, MSG_ORIG(MSG_ARG_REDUCE)) == 0)
                                ofl->ofl_flags |= FLG_OF_PROCRED;
                        else if (strcmp(optarg, MSG_ORIG(MSG_STR_LOCAL)) == 0)
                                Blflag = TRUE;
                        else if (strcmp(optarg, MSG_ORIG(MSG_ARG_GROUP)) == 0)
                                Bgflag = TRUE;
                        else if (strcmp(optarg,
                            MSG_ORIG(MSG_STR_ELIMINATE)) == 0)
                                Beflag = TRUE;
                        else if (strcmp(optarg,
                            MSG_ORIG(MSG_ARG_TRANSLATOR)) == 0) {
                                ld_eprintf(ofl, ERR_WARNING,
                                    MSG_INTL(MSG_ARG_UNSUPPORTED),
                                    MSG_ORIG(MSG_ARG_BTRANSLATOR));
                        } else if (strcmp(optarg,
                            MSG_ORIG(MSG_STR_LD_DYNAMIC)) &&
                            strcmp(optarg, MSG_ORIG(MSG_ARG_STATIC))) {
                                ld_eprintf(ofl, ERR_FATAL,
                                    MSG_INTL(MSG_ARG_ILLEGAL),
                                    MSG_ORIG(MSG_ARG_CB), optarg);
                        }
                        break;

                case ’G’:
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
                        otype = OT_SHARED;
                        break;

                case ’L’:
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
                        break;

                case ’M’:
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
                        if (aplist_append(&(ofl->ofl_maps), optarg,
                            AL_CNT_OFL_MAPFILES) == NULL)
                                return (S_ERROR);
                        break;

                case ’N’:
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
                        break;

                case ’Q’:
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
```

```
                        if ((optarg[0] == ’n’) && (optarg[1] == ’\0’)) {
                                if (Qflag != SET_UNKNOWN)
                                        ld_eprintf(ofl, ERR_WARNING_NF,
                                            MSG_INTL(MSG_ARG_MTONCE),
                                            MSG_ORIG(MSG_ARG_CQ));
                                else
                                        Qflag = SET_FALSE;
                        } else if ((optarg[0] == ’y’) && (optarg[1] == ’\0’)) {
                                if (Qflag != SET_UNKNOWN)
                                        ld_eprintf(ofl, ERR_WARNING_NF,
                                            MSG_INTL(MSG_ARG_MTONCE),
                                            MSG_ORIG(MSG_ARG_CQ));
                                else
                                        Qflag = SET_TRUE;
                        } else {
                                ld_eprintf(ofl, ERR_FATAL,
                                    MSG_INTL(MSG_ARG_ILLEGAL),
                                    MSG_ORIG(MSG_ARG_CQ), optarg);
                        }
                        break;

                case ’S’:
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
                        if (aplist_append(&lib_support, optarg,
                            AL_CNT_SUPPORT) == NULL)
                                return (S_ERROR);
                        break;

                case ’V’:
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
                        if (!Vflag)
                                (void) fprintf(stderr, MSG_ORIG(MSG_STR_STRNL),
                                    ofl->ofl_sgsid);
                        Vflag = TRUE;
                        break;

                case ’Y’:
                        DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, optarg));
                        if (strncmp(optarg, MSG_ORIG(MSG_ARG_LCOM), 2) == 0) {
                                if (Llibdir)
                                        ld_eprintf(ofl, ERR_WARNING_NF,
                                            MSG_INTL(MSG_ARG_MTONCE),
                                            MSG_ORIG(MSG_ARG_CYL));
                                else
                                        Llibdir = optarg + 2;
                        } else if (strncmp(optarg,
                            MSG_ORIG(MSG_ARG_UCOM), 2) == 0) {
                                if (Ulibdir)
                                        ld_eprintf(ofl, ERR_WARNING_NF,
                                            MSG_INTL(MSG_ARG_MTONCE),
                                            MSG_ORIG(MSG_ARG_CYU));
                                else
                                        Ulibdir = optarg + 2;
                        } else if (strncmp(optarg,
                            MSG_ORIG(MSG_ARG_PCOM), 2) == 0) {
                                if (Plibpath)
                                        ld_eprintf(ofl, ERR_WARNING_NF,
                                            MSG_INTL(MSG_ARG_MTONCE),
                                            MSG_ORIG(MSG_ARG_CYP));
                                else
                                        Plibpath = optarg + 2;
                        } else {
                                ld_eprintf(ofl, ERR_FATAL,
                                    MSG_INTL(MSG_ARG_ILLEGAL),
                                    MSG_ORIG(MSG_ARG_CY), optarg);
                        }
```

```
			break;

		case '?':
			DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c, NULL));
			/*
			 * If the option character is '-', we're looking at a
			 * long option which couldn't be translated, display a
			 * more useful error.
			 */
			if (optopt == '-') {
				eprintf(ofl->ofl_lml, ERR_FATAL,
				    MSG_INTL(MSG_ARG_LONG_UNKNOWN),
				    argv[optind-1]);
			} else {
				eprintf(ofl->ofl_lml, ERR_FATAL,
				    MSG_INTL(MSG_ARG_UNKNOWN), optopt);
			}
			(*usage)++;
			break;

		default:
			break;
		}

		/*
		 * Update the argument index for the next getopt() iteration.
		 */
		ndx = optind;
	}
	return (1);
}

/*
 * Parsing options pass2 for
 */
static uintptr_t
parseopt_pass2(Ofl_desc *ofl, int argc, char **argv)
{
	int	c, ndx = optind;

	while ((c = ld_getopt(ofl->ofl_lml, ndx, argc, argv)) != -1) {
		Ifl_desc	*ifl;
		Sym_desc	*sdp;

		switch (c) {
			case 'l':
				DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
				    optarg));
				if (ld_find_library(optarg, ofl) == S_ERROR)
					return (S_ERROR);
				break;
			case 'B':
				DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
				    optarg));
				if (strcmp(optarg,
				    MSG_ORIG(MSG_STR_LD_DYNAMIC)) == 0) {
					if (ofl->ofl_flags & FLG_OF_DYNAMIC)
						ofl->ofl_flags |=
						    FLG_OF_DYNLIBS;
					else {
						ld_eprintf(ofl, ERR_FATAL,
						    MSG_INTL(MSG_ARG_ST_INCOMP),
						    MSG_ORIG(MSG_ARG_BDYNAMIC));
					}
				} else if (strcmp(optarg,
				    MSG_ORIG(MSG_ARG_STATIC)) == 0)
```

```
					ofl->ofl_flags &= ~FLG_OF_DYNLIBS;
				break;
			case 'L':
				DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
				    optarg));
				if (ld_add_libdir(ofl, optarg) == S_ERROR)
					return (S_ERROR);
				break;
			case 'N':
				DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
				    optarg));
				/*
				 * Record DT_NEEDED string
				 */
				if (!(ofl->ofl_flags & FLG_OF_DYNAMIC))
					ld_eprintf(ofl, ERR_FATAL,
					    MSG_INTL(MSG_ARG_ST_INCOMP),
					    MSG_ORIG(MSG_ARG_CN));
				if (((ifl = libld_calloc(1,
				    sizeof (Ifl_desc))) == NULL) ||
				    (aplist_append(&ofl->ofl_sos, ifl,
				    AL_CNT_OFL_LIBS) == NULL))
					return (S_ERROR);

				ifl->ifl_name = MSG_INTL(MSG_STR_COMMAND);
				ifl->ifl_soname = optarg;
				ifl->ifl_flags = (FLG_IF_NEEDSTR |
				    FLG_IF_FILEREF | FLG_IF_DEPREQD);

				break;
			case 'D':
				DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
				    optarg));
				(void) dbg_setup(ofl, optarg, 3);
				break;
			case 'u':
				DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
				    optarg));
				if (ld_sym_add_u(optarg, ofl,
				    MSG_STR_COMMAND) == (Sym_desc *)S_ERROR)
					return (S_ERROR);
				break;
			case 'z':
				DBG_CALL(Dbg_args_option(ofl->ofl_lml, ndx, c,
				    optarg));
				if ((strncmp(optarg, MSG_ORIG(MSG_ARG_LD32),
				    MSG_ARG_LD32_SIZE) == 0) ||
				    (strncmp(optarg, MSG_ORIG(MSG_ARG_LD64),
				    MSG_ARG_LD64_SIZE) == 0)) {
					if (createargv(ofl, 0) == S_ERROR)
						return (S_ERROR);
				} else if (strcmp(optarg,
				    MSG_ORIG(MSG_ARG_ALLEXTRT)) == 0) {
					ofl->ofl_flags1 |= FLG_OF1_ALLEXRT;
					ofl->ofl_flags1 &= ~FLG_OF1_WEAKEXT;
				} else if (strcmp(optarg,
				    MSG_ORIG(MSG_ARG_WEAKEXT)) == 0) {
					ofl->ofl_flags1 |= FLG_OF1_WEAKEXT;
					ofl->ofl_flags1 &= ~FLG_OF1_ALLEXRT;
				} else if (strcmp(optarg,
				    MSG_ORIG(MSG_ARG_DFLEXTRT)) == 0) {
					ofl->ofl_flags1 &=
					    ~(FLG_OF1_ALLEXRT |
					    FLG_OF1_WEAKEXT);
				} else if (strcmp(optarg,
				    MSG_ORIG(MSG_ARG_DIRECT)) == 0) {
```

```
                                        ofl->ofl_flags1 |= FLG_OF1_ZDIRECT;
                                        ofl->ofl_guideflags |= FLG_OFG_NO_DB;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_NODIRECT)) == 0) {
                                        ofl->ofl_flags1 &= ~FLG_OF1_ZDIRECT;
                                        ofl->ofl_guideflags |= FLG_OFG_NO_DB;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_IGNORE)) == 0) {
                                        ofl->ofl_flags1 |= FLG_OF1_IGNORE;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_RECORD)) == 0) {
                                        ofl->ofl_flags1 &= ~FLG_OF1_IGNORE;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_LAZYLOAD)) == 0) {
                                        ofl->ofl_flags1 |= FLG_OF1_LAZYLD;
                                        ofl->ofl_guideflags |= FLG_OFG_NO_LAZY;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_NOLAZYLOAD)) == 0) {
                                        ofl->ofl_flags1 &= ~ FLG_OF1_LAZYLD;
                                        ofl->ofl_guideflags |= FLG_OFG_NO_LAZY;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_GROUPPERM)) == 0) {
                                        ofl->ofl_flags1 |= FLG_OF1_GRPPRM;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_NOGROUPPERM)) == 0) {
                                        ofl->ofl_flags1 &= ~FLG_OF1_GRPPRM;
                                } else if (strncmp(optarg,
                                    MSG_ORIG(MSG_ARG_INITARRAY),
                                    MSG_ARG_INITARRAY_SIZE) == 0) {
                                        if (((sdp = ld_sym_add_u(optarg +
                                            MSG_ARG_INITARRAY_SIZE, ofl,
                                            MSG_STR_COMMAND)) ==
                                            (Sym_desc *)S_ERROR) ||
                                            (aplist_append(&ofl->ofl_initarray,
                                            sdp, AL_CNT_OFL_ARRAYS) == NULL))
                                                return (S_ERROR);
                                } else if (strncmp(optarg,
                                    MSG_ORIG(MSG_ARG_FINIARRAY),
                                    MSG_ARG_FINIARRAY_SIZE) == 0) {
                                        if (((sdp = ld_sym_add_u(optarg +
                                            MSG_ARG_FINIARRAY_SIZE, ofl,
                                            MSG_STR_COMMAND)) ==
                                            (Sym_desc *)S_ERROR) ||
                                            (aplist_append(&ofl->ofl_finiarray,
                                            sdp, AL_CNT_OFL_ARRAYS) == NULL))
                                                return (S_ERROR);
                                } else if (strncmp(optarg,
                                    MSG_ORIG(MSG_ARG_PREINITARRAY),
                                    MSG_ARG_PREINITARRAY_SIZE) == 0) {
                                        if (((sdp = ld_sym_add_u(optarg +
                                            MSG_ARG_PREINITARRAY_SIZE, ofl,
                                            MSG_STR_COMMAND)) ==
                                            (Sym_desc *)S_ERROR) ||
                                            (aplist_append(&ofl->ofl_preiarray,
                                            sdp, AL_CNT_OFL_ARRAYS) == NULL))
                                                return (S_ERROR);
                                } else if (strncmp(optarg,
                                    MSG_ORIG(MSG_ARG_RTLDINFO),
                                    MSG_ARG_RTLDINFO_SIZE) == 0) {
                                        if (((sdp = ld_sym_add_u(optarg +
                                            MSG_ARG_RTLDINFO_SIZE, ofl,
                                            MSG_STR_COMMAND)) ==
                                            (Sym_desc *)S_ERROR) ||
                                            (aplist_append(&ofl->ofl_rtldinfo,
                                            sdp, AL_CNT_OFL_ARRAYS) == NULL))
                                                return (S_ERROR);
```

```
                                } else if (strncmp(optarg,
                                    MSG_ORIG(MSG_ARG_DTRACE),
                                    MSG_ARG_DTRACE_SIZE) == 0) {
                                        if ((sdp = ld_sym_add_u(optarg +
                                            MSG_ARG_DTRACE_SIZE, ofl,
                                            MSG_STR_COMMAND)) ==
                                            (Sym_desc *)S_ERROR)
                                                return (S_ERROR);
                                        ofl->ofl_dtracesym = sdp;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_RESCAN_NOW)) == 0) {
                                        if (ld_rescan_archives(ofl, 0, ndx) ==
                                            S_ERROR)
                                                return (S_ERROR);
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_RESCAN_START)) == 0) {
                                        ofl->ofl_ars_gsndx = ofl->ofl_arscnt;
                                        ofl->ofl_ars_gsandx = ndx;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_RESCAN_END)) == 0) {
                                        if (ld_rescan_archives(ofl, 1, ndx) ==
                                            S_ERROR)
                                                return (S_ERROR);
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_DEFERRED)) == 0) {
                                        ofl->ofl_flags1 |= FLG_OF1_DEFERRED;
                                } else if (strcmp(optarg,
                                    MSG_ORIG(MSG_ARG_NODEFERRED)) == 0) {
                                        ofl->ofl_flags1 &= ~FLG_OF1_DEFERRED;
                                }
                        default:
                                break;
                }

                /*
                 * Update the argument index for the next getopt() iteration.
                 */
                ndx = optind;
        }
        return (1);
}

/*
 *
 * Pass 1 -- process_flags: collects all options and sets flags
 */
static uintptr_t
process_flags_com(Ofl_desc *ofl, int argc, char **argv, int *usage)
{
        for (; optind < argc; optind++) {
                /*
                 * If we detect some more options return to getopt().
                 * Checking argv[optind][1] against null prevents a forever
                 * loop if an unadorned '-' argument is passed to us.
                 */
                while ((optind < argc) && (argv[optind][0] == '-')) {
                        if (argv[optind][1] != '\0') {
                                if (parseopt_pass1(ofl, argc, argv,
                                    usage) == S_ERROR)
                                        return (S_ERROR);
                        } else if (++optind < argc)
                                continue;
                }
                if (optind >= argc)
                        break;
                ofl->ofl_objscnt++;
	}

	/* Did an unterminated archive group run off the end? */
	if (ofl->ofl_ars_gsandx > 0) {
		ld_eprintf(ofl, ERR_FATAL, MSG_INTL(MSG_ARG_AR_GRP_BAD),
		    MSG_INTL(MSG_MARG_AR_GRP_START),
		    MSG_INTL(MSG_MARG_AR_GRP_END));
		return (S_ERROR);
	}

	return (1);
}

uintptr_t
ld_process_flags(Ofl_desc *ofl, int argc, char **argv)
{
	int	usage = 0;	/* Collect all argument errors before exit */

	if (argc < 2) {
		usage_mesg(FALSE);
		return (S_ERROR);
	}

	/*
	 * Option handling
	 */
	opterr = 0;
	optind = 1;
	if (process_flags_com(ofl, argc, argv, &usage) == S_ERROR)
		return (S_ERROR);

	/*
	 * Having parsed everything, did we have any usage errors.
	 */
	if (usage) {
		eprintf(ofl->ofl_lml, ERR_FATAL, MSG_INTL(MSG_ARG_USEHELP));
		return (S_ERROR);
	}

	return (check_flags(ofl, argc));
}

/*
 * Pass 2 -- process_files: skips the flags collected in pass 1 and processes
 * files.
 */
static uintptr_t
process_files_com(Ofl_desc *ofl, int argc, char **argv)
{
	for (; optind < argc; optind++) {
		int		fd;
		uintptr_t	open_ret;
		char		*path;
		Rej_desc	rej = { 0 };

		/*
		 * If we detect some more options return to getopt().
		 * Checking argv[optind][1] against null prevents a forever
		 * loop if an unadorned '-' argument is passed to us.
		 */
		while ((optind < argc) && (argv[optind][0] == '-')) {
			if (argv[optind][1] != '\0') {
				if (parseopt_pass2(ofl, argc, argv) == S_ERROR)
					return (S_ERROR);
			} else if (++optind < argc)
				continue;
```

```
		}
		if (optind >= argc)
			break;

		path = argv[optind];
		if ((fd = open(path, O_RDONLY)) == -1) {
			int err = errno;

			ld_eprintf(ofl, ERR_FATAL,
			    MSG_INTL(MSG_SYS_OPEN), path, strerror(err));
			continue;
		}

		DBG_CALL(Dbg_args_file(ofl->ofl_lml, optind, path));

		open_ret = ld_process_open(path, path, &fd, ofl,
		    (FLG_IF_CMDLINE | FLG_IF_NEEDED), &rej, NULL);
		if (fd != -1)
			(void) close(fd);
		if (open_ret == S_ERROR)
			return (S_ERROR);

		/*
		 * Check for mismatched input.
		 */
		if (rej.rej_type) {
			Conv_reject_desc_buf_t rej_buf;

			ld_eprintf(ofl, ERR_FATAL,
			    MSG_INTL(reject[rej.rej_type]),
			    rej.rej_name ? rej.rej_name :
			    MSG_INTL(MSG_STR_UNKNOWN),
			    conv_reject_desc(&rej, &rej_buf,
			    ld_targ.t_m.m_mach));
			return (1);
		}
	}
	return (1);
}

uintptr_t
ld_process_files(Ofl_desc *ofl, int argc, char **argv)
{
	DBG_CALL(Dbg_basic_files(ofl->ofl_lml));

	/*
	 * Process command line files (taking into account any applicable
	 * preceding flags).  Return if any fatal errors have occurred.
	 */
	opterr = 0;
	optind = 1;
	if (process_files_com(ofl, argc, argv) == S_ERROR)
		return (S_ERROR);
	if (ofl->ofl_flags & FLG_OF_FATAL)
		return (1);

	/*
	 * Guidance: Use -B direct/nodirect or -z direct/nodirect.
	 *
	 * This is a backstop for the case where the link had no dependencies.
	 * Otherwise, it will get caught by ld_process_ifl(). We need both,
	 * because -z direct is positional, and its value at the time where
	 * the first dependency is seen might be different than it is now.
	 */
	if ((ofl->ofl_flags & FLG_OF_DYNAMIC) &&
	    OFL_GUIDANCE(ofl, FLG_OFG_NO_DB)) {
```

```
		ld_eprintf(ofl, ERR_GUIDANCE, MSG_INTL(MSG_GUIDE_DIRECT));
		ofl->ofl_guideflags |= FLG_OFG_NO_DB;
	}

	/*
	 * Now that all command line files have been processed see if there are
	 * any additional `needed' shared object dependencies.
	 */
	if (ofl->ofl_soneed)
		if (ld_finish_libs(ofl) == S_ERROR)
			return (S_ERROR);

	/*
	 * If rescanning archives is enabled, do so now to determine whether
	 * there might still be members extracted to satisfy references from any
	 * explicit objects.  Continue until no new objects are extracted.  Note
	 * that this pass is carried out *after* processing any implicit objects
	 * (above) as they may already have resolved any undefined references
	 * from any explicit dependencies.
	 */
	if (ofl->ofl_flags1 & FLG_OF1_RESCAN) {
		if (ld_rescan_archives(ofl, 0, argc) == S_ERROR)
			return (S_ERROR);
		if (ofl->ofl_flags & FLG_OF_FATAL)
			return (1);
	}

	/*
	 * If debugging, provide statistics on each archives extraction, or flag
	 * any archive that has provided no members.  Note that this could be a
	 * nice place to free up much of the archive infrastructure, as we've
	 * extracted any members we need.  However, as we presently don't free
	 * anything under ld(1) there's not much point in proceeding further.
	 */
	DBG_CALL(Dbg_statistics_ar(ofl));

	/*
	 * If any version definitions have been established, either via input
	 * from a mapfile or from the input relocatable objects, make sure any
	 * version dependencies are satisfied, and version symbols created.
	 */
	if (ofl->ofl_verdesc)
		if (ld_vers_check_defs(ofl) == S_ERROR)
			return (S_ERROR);

	/*
	 * If input section ordering was specified within some segment
	 * using a mapfile, verify that the expected sections were seen.
	 */
	if (ofl->ofl_flags & FLG_OF_IS_ORDER)
		ld_ent_check(ofl);

	return (1);
}

uintptr_t
ld_init_strings(Ofl_desc *ofl)
{
	uint_t	stflags;

	if (ofl->ofl_flags1 & FLG_OF1_NCSTTAB)
		stflags = 0;
	else
		stflags = FLG_STNEW_COMPRESS;

	if (((ofl->ofl_shdrsttab = st_new(stflags)) == NULL) ||
```

```
	    ((ofl->ofl_strtab = st_new(stflags)) == NULL) ||
	    ((ofl->ofl_dynstrtab = st_new(stflags)) == NULL))
		return (S_ERROR);

	return (0);
}
```

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

/*
 * Process the current input file.  There are basically three types of files
 * that come through here:
 *
 *  -   files explicitly defined on the command line (ie. foo.o or bar.so),
 *      in this case only the 'name' field is valid.
 *
 *  -   libraries determined from the -l command line option (ie. -lbar),
 *      in this case the 'soname' field contains the basename of the located
 *      file.
 *
 * Any shared object specified via the above two conventions must be recorded
 * as a needed dependency.
 *
 *  -   libraries specified as dependencies of those libraries already obtained
 *      via the command line (ie. bar.so has a DT_NEEDED entry of fred.so.1),
 *      in this case the 'soname' field contains either a full pathname (if the
 *      needed entry contained a '/'), or the basename of the located file.
 *      These libraries are processed to verify symbol binding but are not
 *      recorded as dependencies of the output file being generated.
 *
 * entry:
 *      name - File name
 *      soname - SONAME for needed sharable library, as described above
 *      fd - Open file descriptor
 *      elf - Open ELF handle
 *      flags - FLG_IF_ flags applicable to file
 *      ofl - Output file descriptor
 *      rej - Rejection descriptor used to record rejection reason
 *      ifl_ret - NULL, or address of pointer to receive reference to
 *              resulting input descriptor for file. If ifl_ret is non-NULL,
 *              the file cannot be an archive or it will be rejected.
 *
 * exit:
 *      If a error occurs in examining the file, S_ERROR is returned.
 *      If the file can be examined, but is not suitable, *rej is updated,
 *      and 0 is returned. If the file is acceptable, 1 is returned, and if
 *      ifl_ret is non-NULL, *ifl_ret is set to contain the pointer to the
 *      resulting input descriptor.
 */
uintptr_t
ld_process_ifl(const char *name, const char *soname, int fd, Elf *elf,
    Word flags, Ofl_desc *ofl, Rej_desc *rej, Ifl_desc **ifl_ret)
{
        Ifl_desc        *ifl;
        Ehdr            *ehdr;
        uintptr_t       error = 0;
        struct stat     status;
        Ar_desc         *adp;
        Rej_desc        _rej;

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

```
        else {
                status.st_dev = 0;
                status.st_ino = 0;
        }

        switch (elf_kind(elf)) {
        case ELF_K_AR:
                /*
                 * If the caller has supplied a non-NULL ifl_ret, then
                 * we cannot process archives, for there will be no
                 * input file descriptor for us to return. In this case,
                 * reject the attempt.
                 */
                if (ifl_ret != NULL) {
                        _rej.rej_type = SGS_REJ_ARCHIVE;
                        _rej.rej_name = name;
                        DBG_CALL(Dbg_file_rejected(ofl->ofl_lml, &_rej,
                            ld_targ.t_m.m_mach));
                        if (rej->rej_type == 0) {
                                *rej = _rej;
                                rej->rej_name = strdup(_rej.rej_name);
                        }
                        return (0);
                }

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

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

                                /*
                                 * We've seen this file before so reuse the
                                 * original archive descriptor and discard the
                                 * new elf descriptor.  Note that a file
                                 * descriptor is unnecessary, as the file is
                                 * already available in memory.
                                 */
                                DBG_CALL(Dbg_file_reuse(ofl->ofl_lml, name,
                                    adp->ad_name));
                                (void) elf_end(elf);
                                if (!ld_process_archive(name, -1, adp, ofl))
                                        return (S_ERROR);
                                return (1);
                        }
                }

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

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

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

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

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

                        /*
                         * This can fail for a number of reasons. Typically
                         * the object class is incorrect (ie. user is building
                         * 64-bit but managed to point at 32-bit libraries).
                         * Other ELF errors can include a truncated or corrupt
                         * file. Try to get the best error message possible.
                         */
                        if (ld_targ.t_m.m_class != _class) {
                                _rej.rej_type = SGS_REJ_CLASS;
                                _rej.rej_info = (uint_t)_class;
                        } else {
                                _rej.rej_type = SGS_REJ_STR;
                                _rej.rej_str = elf_errmsg(-1);
                        }
                        _rej.rej_name = name;
                        DBG_CALL(Dbg_file_rejected(ofl->ofl_lml, &_rej,
                            ld_targ.t_m.m_mach));
                        if (rej->rej_type == 0) {
                                *rej = _rej;
                                rej->rej_name = strdup(_rej.rej_name);
                        }
                        return (0);
                }

                if (_gelf_getdynval(elf, DT_SUNW_KMOD) > 0) {
                if (_gelf_getdynval(elf, DT_SUNW_KMOD) == 1) {
                        _rej.rej_name = name;
                        DBG_CALL(Dbg_file_rejected(ofl->ofl_lml, &_rej,
                            ld_targ.t_m.m_mach));
                        _rej.rej_type = SGS_REJ_KMOD;
                        _rej.rej_str = elf_errmsg(-1);
                        _rej.rej_name = name;
#endif /* ! codereview */

                        if (rej->rej_type == 0) {
                                *rej = _rej;
                                rej->rej_name = strdup(_rej.rej_name);
                        }
```

```
                        return (0);
                }

                /*
                 * Determine if we've already come across this file.
                 */
                if (!(flags & FLG_IF_EXTRACT)) {
                        APlist  *apl;
                        Aliste  idx;

                        if (ehdr->e_type == ET_REL)
                                apl = ofl->ofl_objs;
                        else
                                apl = ofl->ofl_sos;

                        /*
                         * Traverse the appropriate file list and determine if
                         * a dev/inode match is found.
                         */
                        for (APLIST_TRAVERSE(apl, idx, ifl)) {
                                /*
                                 * Ifl_desc generated via -Nneed, therefore no
                                 * actual file behind it.
                                 */
                                if (ifl->ifl_flags & FLG_IF_NEEDSTR)
                                        continue;

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

                                /*
                                 * Disregard (skip) this image.
                                 */
                                DBG_CALL(Dbg_file_skip(ofl->ofl_lml,
                                    ifl->ifl_name, name));
                                (void) elf_end(elf);

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

                                        /*
                                         * Determine whether this is the same
                                         * file name as originally encountered
                                         * so as to provide the most
                                         * descriptive diagnostic.
                                         */
                                        errmsg =
                                            (strcmp(name, ifl->ifl_name) == 0) ?
                                            MSG_INTL(MSG_FIL_MULINC_1) :
                                            MSG_INTL(MSG_FIL_MULINC_2);
                                        ld_eprintf(ofl, ERR_WARNING,
                                            errmsg, name, ifl->ifl_name);
                                }
                                if (ifl_ret)
                                        *ifl_ret = ifl;
                                return (1);
```

```
			}
		}

		/*
		 * At this point, we know we need the file.  Establish an input
		 * file descriptor and continue processing.
		 */
		ifl = ifl_setup(name, ehdr, elf, flags, ofl, rej);
		if ((ifl == NULL) || (ifl == (Ifl_desc *)S_ERROR))
			return ((uintptr_t)ifl);
		ifl->ifl_stdev = status.st_dev;
		ifl->ifl_stino = status.st_ino;

		/*
		 * If -zignore is in effect, mark this file as a potential
		 * candidate (the files use isn't actually determined until
		 * symbol resolution and relocation processing are completed).
		 */
		if (ofl->ofl_flags1 & FLG_OF1_IGNORE)
			ifl->ifl_flags |= FLG_IF_IGNORE;

		switch (ehdr->e_type) {
		case ET_REL:
			(*ld_targ.t_mr.mr_mach_eflags)(ehdr, ofl);
			error = process_elf(ifl, elf, ofl);
			break;
		case ET_DYN:
			if ((ofl->ofl_flags & FLG_OF_STATIC) ||
			    !(ofl->ofl_flags & FLG_OF_DYNLIBS)) {
				ld_eprintf(ofl, ERR_FATAL,
				    MSG_INTL(MSG_FIL_SOINSTAT), name);
				return (0);
			}

			/*
			 * Record any additional shared object information.
			 * If no soname is specified (eg. this file was
			 * derived from a explicit filename declaration on the
			 * command line, ie. bar.so) use the pathname.
			 * This entry may be overridden if the files dynamic
			 * section specifies an DT_SONAME value.
			 */
			if (soname == NULL)
				ifl->ifl_soname = ifl->ifl_name;
			else
				ifl->ifl_soname = soname;

			/*
			 * If direct bindings, lazy loading, group permissions,
			 * or deferred dependencies need to be established, mark
			 * this object.
			 */
			if (ofl->ofl_flags1 & FLG_OF1_ZDIRECT)
				ifl->ifl_flags |= FLG_IF_DIRECT;
			if (ofl->ofl_flags1 & FLG_OF1_LAZYLD)
				ifl->ifl_flags |= FLG_IF_LAZYLD;
			if (ofl->ofl_flags1 & FLG_OF1_GRPPRM)
				ifl->ifl_flags |= FLG_IF_GRPPRM;
			if (ofl->ofl_flags1 & FLG_OF1_DEFERRED)
				ifl->ifl_flags |=
				    (FLG_IF_LAZYLD | FLG_IF_DEFERRED);

			error = process_elf(ifl, elf, ofl);

			/*
			 * Determine whether this dependency requires a syminfo.
```

```
			 */
			if (ifl->ifl_flags & MSK_IF_SYMINFO)
				ofl->ofl_flags |= FLG_OF_SYMINFO;

			/*
			 * Guidance: Use -z lazyload/nolazyload.
			 * libc is exempt from this advice, because it cannot
			 * be lazy loaded, and requests to do so are ignored.
			 */
			if (OFL_GUIDANCE(ofl, FLG_OFG_NO_LAZY) &&
			    ((ifl->ifl_flags & FLG_IF_RTLDINF) == 0)) {
				ld_eprintf(ofl, ERR_GUIDANCE,
				    MSG_INTL(MSG_GUIDE_LAZYLOAD));
				ofl->ofl_guideflags |= FLG_OFG_NO_LAZY;
			}

			/*
			 * Guidance: Use -B direct/nodirect or
			 * -z direct/nodirect.
			 */
			if (OFL_GUIDANCE(ofl, FLG_OFG_NO_DB)) {
				ld_eprintf(ofl, ERR_GUIDANCE,
				    MSG_INTL(MSG_GUIDE_DIRECT));
				ofl->ofl_guideflags |= FLG_OFG_NO_DB;
			}

			break;
		default:
			(void) elf_errno();
			_rej.rej_type = SGS_REJ_UNKFILE;
			_rej.rej_name = name;
			DBG_CALL(Dbg_file_rejected(ofl->ofl_lml, &_rej,
			    ld_targ.t_m.m_mach));
			if (rej->rej_type == 0) {
				*rej = _rej;
				rej->rej_name = strdup(_rej.rej_name);
			}
			return (0);
		}
		break;
	default:
		(void) elf_errno();
		_rej.rej_type = SGS_REJ_UNKFILE;
		_rej.rej_name = name;
		DBG_CALL(Dbg_file_rejected(ofl->ofl_lml, &_rej,
		    ld_targ.t_m.m_mach));
		if (rej->rej_type == 0) {
			*rej = _rej;
			rej->rej_name = strdup(_rej.rej_name);
		}
		return (0);
	}
	if ((error == 0) || (error == S_ERROR))
		return (error);

	if (ifl_ret)
		*ifl_ret = ifl;
	return (1);
}

/*
 * Having successfully opened a file, set up the necessary elf structures to
 * process it further.  This small section of processing is slightly different
 * from the elf initialization required to process a relocatable object from an
 * archive (see libs.c: ld_process_archive()).
 */
```

```
uintptr_t
ld_process_open(const char *opath, const char *ofile, int *fd, Ofl_desc *ofl,
    Word flags, Rej_desc *rej, Ifl_desc **ifl_ret)
{
        Elf             *elf;
        const char      *npath = opath;
        const char      *nfile = ofile;

        if ((elf = elf_begin(*fd, ELF_C_READ, NULL)) == NULL) {
                ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_BEGIN), npath);
                return (0);
        }

        /*
         * Determine whether the support library wishes to process this open.
         * The support library may return:
         *   .  a different ELF descriptor (in which case they should have
         *      closed the original)
         *   .  a different file descriptor (in which case they should have
         *      closed the original)
         *   .  a different path and file name (presumably associated with
         *      a different file descriptor)
         *
         * A file descriptor of -1, or and ELF descriptor of zero indicates
         * the file should be ignored.
         */
        ld_sup_open(ofl, &npath, &nfile, fd, flags, &elf, NULL, 0,
            elf_kind(elf));

        if ((*fd == -1) || (elf == NULL))
                return (0);

        return (ld_process_ifl(npath, nfile, *fd, elf, flags, ofl, rej,
            ifl_ret));
}

/*
 * Having successfully mapped a file, set up the necessary elf structures to
 * process it further.  This routine is patterned after ld_process_open() and
 * is only called by ld.so.1(1) to process a relocatable object.
 */
Ifl_desc *
ld_process_mem(const char *path, const char *file, char *addr, size_t size,
    Ofl_desc *ofl, Rej_desc *rej)
{
        Elf             *elf;
        uintptr_t       open_ret;
        Ifl_desc        *ifl;

        if ((elf = elf_memory(addr, size)) == NULL) {
                ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_MEMORY), path);
                return (0);
        }

        open_ret = ld_process_ifl(path, file, 0, elf, 0, ofl, rej, &ifl);
        if (open_ret != 1)
                return ((Ifl_desc *) open_ret);
        return (ifl);
}

/*
 * Process a required library (i.e. the dependency of a shared object).
 * Combine the directory and filename, check the resultant path size, and try
 * opening the pathname.
 */
static Ifl_desc *
```

```
process_req_lib(Sdf_desc *sdf, const char *dir, const char *file,
    Ofl_desc *ofl, Rej_desc *rej)
{
        size_t          dlen, plen;
        int             fd;
        char            path[PATH_MAX];
        const char      *_dir = dir;

        /*
         * Determine the sizes of the directory and filename to insure we don't
         * exceed our buffer.
         */
        if ((dlen = strlen(dir)) == 0) {
                _dir = MSG_ORIG(MSG_STR_DOT);
                dlen = 1;
        }
        dlen++;
        plen = dlen + strlen(file) + 1;
        if (plen > PATH_MAX) {
                ld_eprintf(ofl, ERR_FATAL, MSG_INTL(MSG_FIL_PTHTOLONG),
                    _dir, file);
                return (0);
        }

        /*
         * Build the entire pathname and try and open the file.
         */
        (void) strcpy(path, _dir);
        (void) strcat(path, MSG_ORIG(MSG_STR_SLASH));
        (void) strcat(path, file);
        DBG_CALL(Dbg_libs_req(ofl->ofl_lml, sdf->sdf_name,
            sdf->sdf_rfile, path));

        if ((fd = open(path, O_RDONLY)) == -1)
                return (0);
        else {
                uintptr_t       open_ret;
                Ifl_desc        *ifl;
                char            *_path;

                if ((_path = libld_malloc(strlen(path) + 1)) == NULL)
                        return ((Ifl_desc *)S_ERROR);
                (void) strcpy(_path, path);
                open_ret = ld_process_open(_path, &_path[dlen], &fd, ofl,
                    0, rej, &ifl);
                if (fd != -1)
                        (void) close(fd);
                if (open_ret != 1)
                        return ((Ifl_desc *)open_ret);
                return (ifl);
        }
}

/*
 * Finish any library processing.  Walk the list of so's that have been listed
 * as "included" by shared objects we have previously processed.  Examine them,
 * without adding them as explicit dependents of this program, in order to
 * complete our symbol definition process.  The search path rules are:
 *
 *  -   use any user supplied paths, i.e. LD_LIBRARY_PATH and -L, then
 *
 *  -   use any RPATH defined within the parent shared object, then
 *
 *  -   use the default directories, i.e. LIBPATH or -YP.
 */
uintptr_t
```

```
ld_finish_libs(Ofl_desc *ofl)
{
        Aliste          idx1;
        Sdf_desc        *sdf;
        Rej_desc        rej = { 0 };

        /*
         * Make sure we are back in dynamic mode.
         */
        ofl->ofl_flags |= FLG_OF_DYNLIBS;

        for (APLIST_TRAVERSE(ofl->ofl_soneed, idx1, sdf)) {
                Aliste          idx2;
                char            *path, *slash = NULL;
                int             fd;
                Ifl_desc        *ifl;
                char            *file = (char *)sdf->sdf_name;

                /*
                 * See if this file has already been processed.  At the time
                 * this implicit dependency was determined there may still have
                 * been more explicit dependencies to process.  Note, if we ever
                 * do parse the command line three times we would be able to
                 * do all this checking when processing the dynamic section.
                 */
                if (sdf->sdf_file)
                        continue;

                for (APLIST_TRAVERSE(ofl->ofl_sos, idx2, ifl)) {
                        if (!(ifl->ifl_flags & FLG_IF_NEEDSTR) &&
                            (strcmp(file, ifl->ifl_soname) == 0)) {
                                sdf->sdf_file = ifl;
                                break;
                        }
                }
                if (sdf->sdf_file)
                        continue;

                /*
                 * If the current path name element embeds a "/", then it's to
                 * be taken "as is", with no searching involved.  Process all
                 * "/" occurrences, so that we can deduce the base file name.
                 */
                for (path = file; *path; path++) {
                        if (*path == '/')
                                slash = path;
                }
                if (slash) {
                        DBG_CALL(Dbg_libs_req(ofl->ofl_lml, sdf->sdf_name,
                            sdf->sdf_rfile, file));
                        if ((fd = open(file, O_RDONLY)) == -1) {
                                ld_eprintf(ofl, ERR_WARNING,
                                    MSG_INTL(MSG_FIL_NOTFOUND), file,
                                    sdf->sdf_rfile);
                        } else {
                                uintptr_t       open_ret;
                                Rej_desc        _rej = { 0 };

                                open_ret = ld_process_open(file, ++slash,
                                    &fd, ofl, 0, &_rej, &ifl);
                                if (fd != -1)
                                        (void) close(fd);
                                if (open_ret == S_ERROR)
                                        return (S_ERROR);

                                if (_rej.rej_type) {
```

```
                                        Conv_reject_desc_buf_t rej_buf;

                                        ld_eprintf(ofl, ERR_WARNING,
                                            MSG_INTL(reject[_rej.rej_type]),
                                            _rej.rej_name ? rej.rej_name :
                                            MSG_INTL(MSG_STR_UNKNOWN),
                                            conv_reject_desc(&_rej, &rej_buf,
                                            ld_targ.t_m.m_mach));
                                } else
                                        sdf->sdf_file = ifl;
                        }
                        continue;
                }

                /*
                 * Now search for this file in any user defined directories.
                 */
                for (APLIST_TRAVERSE(ofl->ofl_ulibdirs, idx2, path)) {
                        Rej_desc        _rej = { 0 };

                        ifl = process_req_lib(sdf, path, file, ofl, &_rej);
                        if (ifl == (Ifl_desc *)S_ERROR) {
                                return (S_ERROR);
                        }
                        if (_rej.rej_type) {
                                if (rej.rej_type == 0) {
                                        rej = _rej;
                                        rej.rej_name = strdup(_rej.rej_name);
                                }
                        }
                        if (ifl) {
                                sdf->sdf_file = ifl;
                                break;
                        }
                }
                if (sdf->sdf_file)
                        continue;

                /*
                 * Next use the local rules defined within the parent shared
                 * object.
                 */
                if (sdf->sdf_rpath != NULL) {
                        char    *rpath, *next;

                        rpath = libld_malloc(strlen(sdf->sdf_rpath) + 1);
                        if (rpath == NULL)
                                return (S_ERROR);
                        (void) strcpy(rpath, sdf->sdf_rpath);
                        DBG_CALL(Dbg_libs_path(ofl->ofl_lml, rpath,
                            LA_SER_RUNPATH, sdf->sdf_rfile));
                        if ((path = strtok_r(rpath,
                            MSG_ORIG(MSG_STR_COLON), &next)) != NULL) {
                                do {
                                        Rej_desc        _rej = { 0 };

                                        path = expand(sdf->sdf_rfile, path,
                                            &next);

                                        ifl = process_req_lib(sdf, path,
                                            file, ofl, &_rej);
                                        if (ifl == (Ifl_desc *)S_ERROR) {
                                                return (S_ERROR);
                                        }
                                        if ((_rej.rej_type) &&
                                            (rej.rej_type == 0)) {
```

```
						rej = _rej;
						rej.rej_name =
						    strdup(_rej.rej_name);
					}
					if (ifl) {
						sdf->sdf_file = ifl;
						break;
					}
				} while ((path = strtok_r(NULL,
				    MSG_ORIG(MSG_STR_COLON), &next)) != NULL);
			}
		}
		if (sdf->sdf_file)
			continue;

		/*
		 * Finally try the default library search directories.
		 */
		for (APLIST_TRAVERSE(ofl->ofl_dlibdirs, idx2, path)) {
			Rej_desc	_rej = { 0 };

			ifl = process_req_lib(sdf, path, file, ofl, &rej);
			if (ifl == (Ifl_desc *)S_ERROR) {
				return (S_ERROR);
			}
			if (_rej.rej_type) {
				if (rej.rej_type == 0) {
					rej = _rej;
					rej.rej_name = strdup(_rej.rej_name);
				}
			}
			if (ifl) {
				sdf->sdf_file = ifl;
				break;
			}
		}
		if (sdf->sdf_file)
			continue;

		/*
		 * If we've got this far we haven't found the shared object.
		 * If an object was found, but was rejected for some reason,
		 * print a diagnostic to that effect, otherwise generate a
		 * generic "not found" diagnostic.
		 */
		if (rej.rej_type) {
			Conv_reject_desc_buf_t rej_buf;

			ld_eprintf(ofl, ERR_WARNING,
			    MSG_INTL(reject[rej.rej_type]),
			    rej.rej_name ? rej.rej_name :
			    MSG_INTL(MSG_STR_UNKNOWN),
			    conv_reject_desc(&rej, &rej_buf,
			    ld_targ.t_m.m_mach));
		} else {
			ld_eprintf(ofl, ERR_WARNING,
			    MSG_INTL(MSG_FIL_NOTFOUND), file, sdf->sdf_rfile);
		}
	}

	/*
	 * Finally, now that all objects have been input, make sure any version
	 * requirements have been met.
	 */
	return (ld_vers_verify(ofl));
}
```

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

```
/*
 * Make the dynamic section.  Calculate the size of any strings referenced
 * within this structure, they will be added to the global string table
 * (.dynstr).  This routine should be called before make_dynstr().
 *
 * This routine must be maintained in parallel with update_odynamic()
 * in update.c
 */
static uintptr_t
make_dynamic(Ofl_desc *ofl)
{
        Shdr            *shdr;
        Os_desc         *osp;
        Elf_Data        *data;
        Is_desc         *isec;
        size_t          cnt = 0;
        Aliste          idx;
        Ifl_desc        *ifl;
        Sym_desc        *sdp;
        size_t          size;
        Str_tbl         *strtbl;
        ofl_flag_t      flags = ofl->ofl_flags;
        int             not_relobj = !(flags & FLG_OF_RELOBJ);
        int             unused = 0;

        /*
         * Select the required string table.
         */
        if (OFL_IS_STATIC_OBJ(ofl))
                strtbl = ofl->ofl_strtab;
        else
                strtbl = ofl->ofl_dynstrtab;

        /*
         * Only a limited subset of DT_ entries apply to relocatable
         * objects. See the comment at the head of update_odynamic() in
         * update.c for details.
         */
        if (new_section(ofl, SHT_DYNAMIC, MSG_ORIG(MSG_SCN_DYNAMIC), 0,
            &isec, &shdr, &data) == S_ERROR)
                return (S_ERROR);

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

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

        osp = ofl->ofl_osdynamic =
```

```
            ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_dynamic, NULL);

        /*
         * Reserve entries for any needed dependencies.
         */
        for (APLIST_TRAVERSE(ofl->ofl_sos, idx, ifl)) {
                if (!(ifl->ifl_flags & (FLG_IF_NEEDED | FLG_IF_NEEDSTR)))
                        continue;

                /*
                 * If this dependency didn't satisfy any symbol references,
                 * generate a debugging diagnostic (ld(1) -Dunused can be used
                 * to display these).  If this is a standard needed dependency,
                 * and -z ignore is in effect, drop the dependency.  Explicitly
                 * defined dependencies (i.e., -N dep) don't get dropped, and
                 * are flagged as being required to simplify update_odynamic()
                 * processing.
                 */
                if ((ifl->ifl_flags & FLG_IF_NEEDSTR) ||
                    ((ifl->ifl_flags & FLG_IF_DEPREQD) == 0)) {
                        if (unused++ == 0)
                                DBG_CALL(Dbg_util_nl(ofl->ofl_lml, DBG_NL_STD));
                        DBG_CALL(Dbg_unused_file(ofl->ofl_lml, ifl->ifl_soname,
                            (ifl->ifl_flags & FLG_IF_NEEDSTR), 0));

                        /*
                         * Guidance: Remove unused dependency.
                         *
                         * If -z ignore is in effect, this warning is not
                         * needed because we will quietly remove the unused
                         * dependency.
                         */
                        if (OFL_GUIDANCE(ofl, FLG_OFG_NO_UNUSED) &&
                            ((ifl->ifl_flags & FLG_IF_IGNORE) == 0))
                                ld_eprintf(ofl, ERR_GUIDANCE,
                                    MSG_INTL(MSG_GUIDE_UNUSED),
                                    ifl->ifl_soname);

                        if (ifl->ifl_flags & FLG_IF_NEEDSTR)
                                ifl->ifl_flags |= FLG_IF_DEPREQD;
                        else if (ifl->ifl_flags & FLG_IF_IGNORE)
                                continue;
                }

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

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

                /*
                 * If the needed entry contains the $ORIGIN token make sure
                 * the associated DT_1_FLAGS entry is created.
                 */
                if (strstr(ifl->ifl_soname, MSG_ORIG(MSG_STR_ORIGIN))) {
                        ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
                        ofl->ofl_dtflags |= DF_ORIGIN;
                }
        }

        if (unused)
                DBG_CALL(Dbg_util_nl(ofl->ofl_lml, DBG_NL_STD));
```

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

		/*
		 * Reserve entries for _init() and _fini() section addresses.
		 */
		if (((sdp = ld_sym_find(MSG_ORIG(MSG_SYM_INIT_U),
		    SYM_NOHASH, NULL, ofl)) != NULL) &&
		    (sdp->sd_ref == REF_REL_NEED) &&
		    (sdp->sd_sym->st_shndx != SHN_UNDEF)) {
			sdp->sd_flags |= FLG_SY_UPREQD;
			cnt++;
		}
		if (((sdp = ld_sym_find(MSG_ORIG(MSG_SYM_FINI_U),
		    SYM_NOHASH, NULL, ofl)) != NULL) &&
		    (sdp->sd_ref == REF_REL_NEED) &&
		    (sdp->sd_sym->st_shndx != SHN_UNDEF)) {
			sdp->sd_flags |= FLG_SY_UPREQD;
			cnt++;
		}

		/*
		 * Reserve entries for any soname, filter name (shared libs
		 * only), run-path pointers, cache names and audit requirements.
		 */
		if (ofl->ofl_soname) {
			cnt++;
			if (st_insert(strtbl, ofl->ofl_soname) == -1)
				return (S_ERROR);
		}
		if (ofl->ofl_filtees) {
			cnt++;
			if (st_insert(strtbl, ofl->ofl_filtees) == -1)
				return (S_ERROR);

			/*
			 * If the filtees entry contains the $ORIGIN token
			 * make sure the associated DT_1_FLAGS entry is created.
			 */
			if (strstr(ofl->ofl_filtees,
			    MSG_ORIG(MSG_STR_ORIGIN))) {
				ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
				ofl->ofl_dtflags |= DF_ORIGIN;
			}
		}
	}

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

		/*
		 * If the rpath entry contains the $ORIGIN token make sure
		 * the associated DT_1_FLAGS entry is created.
		 */
		if (strstr(ofl->ofl_rpath, MSG_ORIG(MSG_STR_ORIGIN))) {
			ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
			ofl->ofl_dtflags |= DF_ORIGIN;
		}
	}
```

```
	if (not_relobj) {
		Aliste	idx;
		Sg_desc *sgp;

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

			/*
			 * If the config entry contains the $ORIGIN token
			 * make sure the associated DT_1_FLAGS entry is created.
			 */
			if (strstr(ofl->ofl_config, MSG_ORIG(MSG_STR_ORIGIN))) {
				ofl->ofl_dtflags_1 |= DF_1_ORIGIN;
				ofl->ofl_dtflags |= DF_ORIGIN;
			}
		}
		if (ofl->ofl_depaudit) {
			cnt++;
			if (st_insert(strtbl, ofl->ofl_depaudit) == -1)
				return (S_ERROR);
		}
		if (ofl->ofl_audit) {
			cnt++;
			if (st_insert(strtbl, ofl->ofl_audit) == -1)
				return (S_ERROR);
		}

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

		/*
		 * If we are including local functions at the head of
		 * the dynsym, then also reserve entries for DT_SUNW_SYMTAB
		 * and DT_SUNW_SYMSZ.
		 */
		if (OFL_ALLOW_LDYNSYM(ofl))
			cnt += 2;

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

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

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

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

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

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

		if (flags & FLG_OF_TEXTREL)	/* DT_TEXTREL */
			cnt++;
		if (ofl->ofl_osfiniarray)	/* DT_FINI_ARRAY */
			cnt += 2;		/*    DT_FINI_ARRAYSZ */

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

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

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

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

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

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

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

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

		/*
		 * Reserve a entry for each '-zrtldinfo=...' specified
		 * on the command line.
		 */
		for (APLIST_TRAVERSE(ofl->ofl_rtldinfo, idx, sdp))
			cnt++;

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

		/*
		 * Capabilities require a .dynamic entry for the .SUNW_cap
```

```
		 * section.
		 */
		if (ofl->ofl_oscap)
			cnt++;			/* DT_SUNW_CAP */

		/*
		 * Symbol capabilities require a .dynamic entry for the
		 * .SUNW_capinfo section.
		 */
		if (ofl->ofl_oscapinfo)
			cnt++;			/* DT_SUNW_CAPINFO */

		/*
		 * Capabilities chain information requires a .SUNW_capchain
		 * entry (DT_SUNW_CAPCHAIN), entry size (DT_SUNW_CAPCHAINENT),
		 * and total size (DT_SUNW_CAPCHAINSZ).
		 */
		if (ofl->ofl_oscapchain)
			cnt += 3;

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

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

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

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

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

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

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

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

	/*
	 * There are several tags that are specific to the Solaris osabi
	 * range which we unconditionally put into any dynamic section
	 * we create (e.g. DT_SUNW_STRPAD or DT_SUNW_LDMACH). As such,
	 * any Solaris object with a dynamic section should be tagged as
	 * ELFOSABI_SOLARIS.
	 */
	ofl->ofl_flags |= FLG_OF_OSABI;

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

/*
 * Build the GOT section and its associated relocation entries.
 */
uintptr_t
ld_make_got(Ofl_desc *ofl)
{
	Elf_Data	*data;
	Shdr	*shdr;
	Is_desc	*isec;
	size_t	size = (size_t)ofl->ofl_gotcnt * ld_targ.t_m.m_got_entsize;
	size_t	rsize = (size_t)ofl->ofl_relocgotsz;

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

	data->d_size = size;

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

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

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

	return (1);
}

/*
 * Build an interpreter section.
 */
static uintptr_t
make_interp(Ofl_desc *ofl)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;
	const char	*iname = ofl->ofl_interp;
	size_t		size;

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

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

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

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

	size = strlen(iname) + 1;

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

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

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

/*
 * Common function used to build the SHT_SUNW_versym section, SHT_SUNW_syminfo
 * section, and SHT_SUNW_capinfo section.  Each of these sections provide
 * additional symbol information, and their size parallels the associated
 * symbol table.
 */
static Os_desc *
make_sym_sec(Ofl_desc *ofl, const char *sectname, Word stype, int ident)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;

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

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

/*
 * Determine whether a symbol capability is redundant because the object
 * capabilities are more restrictive.
 */
inline static int
is_cap_redundant(Objcapset *ocapset, Objcapset *scapset)
{
	Alist		*oalp, *salp;
	elfcap_mask_t	omsk, smsk;

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

```
         * capabilities will never execute (as the object wouldn’t have been
         * loaded).
         */
        oalp = ocapset->oc_plat.cl_val;
        salp = scapset->oc_plat.cl_val;
        if (oalp && ((salp == NULL) || cap_names_match(oalp, salp)))
                return (1);

        /*
         * If the symbol capability set defines platforms, and the object
         * doesn’t, then the symbol set is more restrictive.
         */
        if (salp && (oalp == NULL))
                return (0);

        /*
         * Next, inspect any machine name capabilities.  If the object defines
         * machine name capabilities, then the object will only be loaded for
         * those machines.  A symbol capability set that doesn’t define the same
         * machine names is redundant, and a symbol capability that does not
         * provide at least one machine name that matches a machine name in the
         * object capabilities will never execute (as the object wouldn’t have
         * been loaded).
         */
        oalp = ocapset->oc_plat.cl_val;
        salp = scapset->oc_plat.cl_val;
        if (oalp && ((salp == NULL) || cap_names_match(oalp, salp)))
                return (1);

        /*
         * If the symbol capability set defines machine names, and the object
         * doesn’t, then the symbol set is more restrictive.
         */
        if (salp && (oalp == NULL))
                return (0);

        /*
         * Next, inspect any hardware capabilities.  If the objects hardware
         * capabilities are greater than or equal to that of the symbols
         * capabilities, then the symbol capability set is redundant.  If the
         * symbols hardware capabilities are greater that the objects, then the
         * symbol set is more restrictive.
         *
         * Note that this is a somewhat arbitrary definition, as each capability
         * bit is independent of the others, and some of the higher order bits
         * could be considered to be less important than lower ones.  However,
         * this is the only reasonable non-subjective definition.
         */
        omsk = ocapset->oc_hw_2.cm_val;
        smsk = scapset->oc_hw_2.cm_val;
        if ((omsk > smsk) || (omsk && (omsk == smsk)))
                return (1);
        if (omsk < smsk)
                return (0);

        /*
         * Finally, inspect the remaining hardware capabilities.
         */
        omsk = ocapset->oc_hw_1.cm_val;
        smsk = scapset->oc_hw_1.cm_val;
        if ((omsk > smsk) || (omsk && (omsk == smsk)))
                return (1);

        return (0);
}
```

```
/*
 * Capabilities values might have been assigned excluded values.  These
 * excluded values should be removed before calculating any capabilities
 * sections size.
 */
static void
capmask_value(Lm_list *lml, Word type, Capmask *capmask, int *title)
{
        /*
         * First determine whether any bits should be excluded.
         */
        if ((capmask->cm_val & capmask->cm_exc) == 0)
                return;

        DBG_CALL(Dbg_cap_post_title(lml, title));

        DBG_CALL(Dbg_cap_val_entry(lml, DBG_STATE_CURRENT, type,
            capmask->cm_val, ld_targ.t_m.m_mach));
        DBG_CALL(Dbg_cap_val_entry(lml, DBG_STATE_EXCLUDE, type,
            capmask->cm_exc, ld_targ.t_m.m_mach));

        capmask->cm_val &= ~capmask->cm_exc;

        DBG_CALL(Dbg_cap_val_entry(lml, DBG_STATE_RESOLVED, type,
            capmask->cm_val, ld_targ.t_m.m_mach));
}

static void
capstr_value(Lm_list *lml, Word type, Caplist *caplist, int *title)
{
        Aliste  idx1, idx2;
        char    *estr;
        Capstr  *capstr;
        Boolean found = FALSE;

        /*
         * First determine whether any strings should be excluded.
         */
        for (APLIST_TRAVERSE(caplist->cl_exc, idx1, estr)) {
                for (ALIST_TRAVERSE(caplist->cl_val, idx2, capstr)) {
                        if (strcmp(estr, capstr->cs_str) == 0) {
                                found = TRUE;
                                break;
                        }
                }
        }

        if (found == FALSE)
                return;

        /*
         * Traverse the current strings, then delete the excluded strings,
         * and finally display the resolved strings.
         */
        if (DBG_ENABLED) {
                Dbg_cap_post_title(lml, title);
                for (ALIST_TRAVERSE(caplist->cl_val, idx2, capstr)) {
                        Dbg_cap_ptr_entry(lml, DBG_STATE_CURRENT, type,
                            capstr->cs_str);
                }
        }
        for (APLIST_TRAVERSE(caplist->cl_exc, idx1, estr)) {
                for (ALIST_TRAVERSE(caplist->cl_val, idx2, capstr)) {
                        if (strcmp(estr, capstr->cs_str) == 0) {
                                DBG_CALL(Dbg_cap_ptr_entry(lml,
                                    DBG_STATE_EXCLUDE, type, capstr->cs_str));
```

```
				alist_delete(caplist->cl_val, &idx2);
				break;
			}
		}
	}
	if (DBG_ENABLED) {
		for (ALIST_TRAVERSE(caplist->cl_val, idx2, capstr)) {
			Dbg_cap_ptr_entry(lml, DBG_STATE_RESOLVED, type,
			    capstr->cs_str);
		}
	}
}

/*
 * Build a capabilities section.
 */
#define	CAP_UPDATE(cap, capndx, tag, val)	\
	cap->c_tag = tag; \
	cap->c_un.c_val = val; \
	cap++, capndx++;

static uintptr_t
make_cap(Ofl_desc *ofl, Word shtype, const char *shname, int ident)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;
	Cap		*cap;
	size_t		size = 0;
	Word		capndx = 0;
	Str_tbl		*strtbl;
	Objcapset	*ocapset = &ofl->ofl_ocapset;
	Aliste		idx1;
	Capstr		*capstr;
	int		title = 0;

	/*
	 * Determine which string table to use for any CA_SUNW_MACH,
	 * CA_SUNW_PLAT, or CA_SUNW_ID strings.
	 */
	if (OFL_IS_STATIC_OBJ(ofl))
		strtbl = ofl->ofl_strtab;
	else
		strtbl = ofl->ofl_dynstrtab;

	/*
	 * If symbol capabilities have been requested, but none have been
	 * created, warn the user.  This scenario can occur if none of the
	 * input relocatable objects defined any object capabilities.
	 */
	if ((ofl->ofl_flags & FLG_OF_OTOSCAP) && (ofl->ofl_capsymcnt == 0))
		ld_eprintf(ofl, ERR_WARNING, MSG_INTL(MSG_CAP_NOSYMSFOUND));

	/*
	 * If symbol capabilities have been collected, but no symbols are left
	 * referencing these capabilities, promote the capability groups back
	 * to an object capability definition.
	 */
	if ((ofl->ofl_flags & FLG_OF_OTOSCAP) && ofl->ofl_capsymcnt &&
	    (ofl->ofl_capfamilies == NULL)) {
		ld_eprintf(ofl, ERR_WARNING, MSG_INTL(MSG_CAP_NOSYMSFOUND));
		ld_cap_move_symtoobj(ofl);
		ofl->ofl_capsymcnt = 0;
		ofl->ofl_capgroups = NULL;
		ofl->ofl_flags &= ~FLG_OF_OTOSCAP;
	}
```

```
	/*
	 * Remove any excluded capabilities.
	 */
	capstr_value(ofl->ofl_lml, CA_SUNW_PLAT, &ocapset->oc_plat, &title);
	capstr_value(ofl->ofl_lml, CA_SUNW_MACH, &ocapset->oc_mach, &title);
	capmask_value(ofl->ofl_lml, CA_SUNW_HW_2, &ocapset->oc_hw_2, &title);
	capmask_value(ofl->ofl_lml, CA_SUNW_HW_1, &ocapset->oc_hw_1, &title);
	capmask_value(ofl->ofl_lml, CA_SUNW_SF_1, &ocapset->oc_sf_1, &title);

	/*
	 * Determine how many entries are required for any object capabilities.
	 */
	size += alist_nitems(ocapset->oc_plat.cl_val);
	size += alist_nitems(ocapset->oc_mach.cl_val);
	if (ocapset->oc_hw_2.cm_val)
		size++;
	if (ocapset->oc_hw_1.cm_val)
		size++;
	if (ocapset->oc_sf_1.cm_val)
		size++;

	/*
	 * Only identify a capabilities group if the group has content.  If a
	 * capabilities identifier exists, and no other capabilities have been
	 * supplied, remove the identifier.  This scenario could exist if a
	 * user mistakenly defined a lone identifier, or if an identified group
	 * was overridden so as to clear the existing capabilities and the
	 * identifier was not also cleared.
	 */
	if (ocapset->oc_id.cs_str) {
		if (size)
			size++;
		else
			ocapset->oc_id.cs_str = NULL;
	}
	if (size)
		size++;			/* Add CA_SUNW_NULL */

	/*
	 * Determine how many entries are required for any symbol capabilities.
	 */
	if (ofl->ofl_capsymcnt) {
		/*
		 * If there are no object capabilities, a CA_SUNW_NULL entry
		 * is required before any symbol capabilities.
		 */
		if (size == 0)
			size++;
		size += ofl->ofl_capsymcnt;
	}

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

	if (new_section(ofl, shtype, shname, size, &isec,
	    &shdr, &data) == S_ERROR)
		return (S_ERROR);

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

	cap = (Cap *)data->d_buf;

	/*
	 * Fill in any object capabilities.  If there is an identifier, then the
	 * identifier comes first.  The remaining items follow in precedence
	 * order, although the order isn't important for runtime verification.
	 */
	if (ocapset->oc_id.cs_str) {
		ofl->ofl_flags |= FLG_OF_CAPSTRS;
		if (st_insert(strtbl, ocapset->oc_id.cs_str) == -1)
			return (S_ERROR);
		ocapset->oc_id.cs_ndx = capndx;
		CAP_UPDATE(cap, capndx, CA_SUNW_ID, 0);
	}
	if (ocapset->oc_plat.cl_val) {
		ofl->ofl_flags |= (FLG_OF_PTCAP | FLG_OF_CAPSTRS);

		/*
		 * Insert any platform name strings in the appropriate string
		 * table.  The capability value can't be filled in yet, as the
		 * final offset of the strings isn't known until later.
		 */
		for (ALIST_TRAVERSE(ocapset->oc_plat.cl_val, idx1, capstr)) {
			if (st_insert(strtbl, capstr->cs_str) == -1)
				return (S_ERROR);
			capstr->cs_ndx = capndx;
			CAP_UPDATE(cap, capndx, CA_SUNW_PLAT, 0);
		}
	}
	if (ocapset->oc_mach.cl_val) {
		ofl->ofl_flags |= (FLG_OF_PTCAP | FLG_OF_CAPSTRS);

		/*
		 * Insert the machine name strings in the appropriate string
		 * table.  The capability value can't be filled in yet, as the
		 * final offset of the strings isn't known until later.
		 */
		for (ALIST_TRAVERSE(ocapset->oc_mach.cl_val, idx1, capstr)) {
			if (st_insert(strtbl, capstr->cs_str) == -1)
				return (S_ERROR);
			capstr->cs_ndx = capndx;
			CAP_UPDATE(cap, capndx, CA_SUNW_MACH, 0);
		}
	}
	if (ocapset->oc_hw_2.cm_val) {
		ofl->ofl_flags |= FLG_OF_PTCAP;
		CAP_UPDATE(cap, capndx, CA_SUNW_HW_2, ocapset->oc_hw_2.cm_val);
	}
	if (ocapset->oc_hw_1.cm_val) {
		ofl->ofl_flags |= FLG_OF_PTCAP;
		CAP_UPDATE(cap, capndx, CA_SUNW_HW_1, ocapset->oc_hw_1.cm_val);
	}
	if (ocapset->oc_sf_1.cm_val) {
		ofl->ofl_flags |= FLG_OF_PTCAP;
		CAP_UPDATE(cap, capndx, CA_SUNW_SF_1, ocapset->oc_sf_1.cm_val);
	}
	CAP_UPDATE(cap, capndx, CA_SUNW_NULL, 0);

	/*
	 * Fill in any symbol capabilities.
	 */
	if (ofl->ofl_capgroups) {
		Cap_group	*cgp;

		for (APLIST_TRAVERSE(ofl->ofl_capgroups, idx1, cgp)) {
			Objcapset	*scapset = &cgp->cg_set;
			Aliste		idx2;
			Is_desc		*isp;

			cgp->cg_ndx = capndx;
```

```
			if (scapset->oc_id.cs_str) {
				ofl->ofl_flags |= FLG_OF_CAPSTRS;
				/*
				 * Insert the identifier string in the
				 * appropriate string table.  The capability
				 * value can't be filled in yet, as the final
				 * offset of the string isn't known until later.
				 */
				if (st_insert(strtbl,
				    scapset->oc_id.cs_str) == -1)
					return (S_ERROR);
				scapset->oc_id.cs_ndx = capndx;
				CAP_UPDATE(cap, capndx, CA_SUNW_ID, 0);
			}

			if (scapset->oc_plat.cl_val) {
				ofl->ofl_flags |= FLG_OF_CAPSTRS;

				/*
				 * Insert the platform name string in the
				 * appropriate string table.  The capability
				 * value can't be filled in yet, as the final
				 * offset of the string isn't known until later.
				 */
				for (ALIST_TRAVERSE(scapset->oc_plat.cl_val,
				    idx2, capstr)) {
					if (st_insert(strtbl,
					    capstr->cs_str) == -1)
						return (S_ERROR);
					capstr->cs_ndx = capndx;
					CAP_UPDATE(cap, capndx,
					    CA_SUNW_PLAT, 0);
				}
			}
			if (scapset->oc_mach.cl_val) {
				ofl->ofl_flags |= FLG_OF_CAPSTRS;

				/*
				 * Insert the machine name string in the
				 * appropriate string table.  The capability
				 * value can't be filled in yet, as the final
				 * offset of the string isn't known until later.
				 */
				for (ALIST_TRAVERSE(scapset->oc_mach.cl_val,
				    idx2, capstr)) {
					if (st_insert(strtbl,
					    capstr->cs_str) == -1)
						return (S_ERROR);
					capstr->cs_ndx = capndx;
					CAP_UPDATE(cap, capndx,
					    CA_SUNW_MACH, 0);
				}
			}
			if (scapset->oc_hw_2.cm_val) {
				CAP_UPDATE(cap, capndx, CA_SUNW_HW_2,
				    scapset->oc_hw_2.cm_val);
			}
			if (scapset->oc_hw_1.cm_val) {
				CAP_UPDATE(cap, capndx, CA_SUNW_HW_1,
				    scapset->oc_hw_1.cm_val);
			}
			if (scapset->oc_sf_1.cm_val) {
				CAP_UPDATE(cap, capndx, CA_SUNW_SF_1,
				    scapset->oc_sf_1.cm_val);
			}
			CAP_UPDATE(cap, capndx, CA_SUNW_NULL, 0);

			/*
			 * If any object capabilities are available, determine
			 * whether these symbol capabilities are less
			 * restrictive, and hence redundant.
			 */
			if (((ofl->ofl_flags & FLG_OF_PTCAP) == 0) ||
			    (is_cap_redundant(ocapset, scapset) == 0))
				continue;

			/*
			 * Indicate any files that provide redundant symbol
			 * capabilities.
			 */
			for (APLIST_TRAVERSE(cgp->cg_secs, idx2, isp)) {
				ld_eprintf(ofl, ERR_WARNING,
				    MSG_INTL(MSG_CAP_REDUNDANT),
				    isp->is_file->ifl_name,
				    EC_WORD(isp->is_scnndx), isp->is_name);
			}
		}
	}

	/*
	 * If capabilities strings are required, the sh_info field of the
	 * section header will be set to the associated string table.
	 */
	if (ofl->ofl_flags & FLG_OF_CAPSTRS)
		shdr->sh_flags |= SHF_INFO_LINK;

	/*
	 * Place these capabilities in the output file.
	 */
	if ((ofl->ofl_oscap = ld_place_section(ofl, isec,
	    NULL, ident, NULL)) == (Os_desc *)S_ERROR)
		return (S_ERROR);

	/*
	 * If symbol capabilities are required, then a .SUNW_capinfo section is
	 * also created.  This table will eventually be sized to match the
	 * associated symbol table.
	 */
	if (ofl->ofl_capfamilies) {
		if ((ofl->ofl_oscapinfo = make_sym_sec(ofl,
		    MSG_ORIG(MSG_SCN_SUNWCAPINFO), SHT_SUNW_capinfo,
		    ld_targ.t_id.id_capinfo)) == (Os_desc *)S_ERROR)
			return (S_ERROR);

		/*
		 * If we’re generating a dynamic object, capabilities family
		 * members are maintained in a .SUNW_capchain section.
		 */
		if (ofl->ofl_capchaincnt &&
		    ((ofl->ofl_flags & FLG_OF_RELOBJ) == 0)) {
			if (new_section(ofl, SHT_SUNW_capchain,
			    MSG_ORIG(MSG_SCN_SUNWCAPCHAIN),
			    ofl->ofl_capchaincnt, &isec, &shdr,
			    &data) == S_ERROR)
				return (S_ERROR);

			ofl->ofl_oscapchain = ld_place_section(ofl, isec,
			    NULL, ld_targ.t_id.id_capchain, NULL);
			if (ofl->ofl_oscapchain == (Os_desc *)S_ERROR)
				return (S_ERROR);
```

```
			}
		}
		return (1);
}
#undef	CAP_UPDATE

/*
 * Build the PLT section and its associated relocation entries.
 */
static uintptr_t
make_plt(Ofl_desc *ofl)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;
	size_t		size = ld_targ.t_m.m_plt_reservsz +
	    (((size_t)ofl->ofl_pltcnt + (size_t)ofl->ofl_pltpad) *
	    ld_targ.t_m.m_plt_entsize);
	size_t		rsize = (size_t)ofl->ofl_relocpltsz;

	/*
	 * On sparc, account for the NOP at the end of the plt.
	 */
	if (ld_targ.t_m.m_mach == LD_TARG_BYCLASS(EM_SPARC, EM_SPARCV9))
		size += sizeof (Word);

	if (new_section(ofl, SHT_PROGBITS, MSG_ORIG(MSG_SCN_PLT), 0,
	    &isec, &shdr, &data) == S_ERROR)
		return (S_ERROR);

	data->d_size = size;
	data->d_align = ld_targ.t_m.m_plt_align;

	shdr->sh_flags = ld_targ.t_m.m_plt_shf_flags;
	shdr->sh_size = (Xword)size;
	shdr->sh_addralign = ld_targ.t_m.m_plt_align;
	shdr->sh_entsize = ld_targ.t_m.m_plt_entsize;

	ofl->ofl_osplt = ld_place_section(ofl, isec, NULL,
	    ld_targ.t_id.id_plt, NULL);
	if (ofl->ofl_osplt == (Os_desc *)S_ERROR)
		return (S_ERROR);

	ofl->ofl_osplt->os_szoutrels = (Xword)rsize;

	return (1);
}

/*
 * Make the hash table.  Only built for dynamic executables and shared
 * libraries, and provides hashed lookup into the global symbol table
 * (.dynsym) for the run-time linker to resolve symbol lookups.
 */
static uintptr_t
make_hash(Ofl_desc *ofl)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;
	size_t		size;
	Word		nsyms = ofl->ofl_globcnt;
	size_t		cnt;

	/*
	 * Allocate section header structures. We set entcnt to 0
	 * because it’s going to change after we place this section.
	 */
	if (new_section(ofl, SHT_HASH, MSG_ORIG(MSG_SCN_HASH), 0,
	    &isec, &shdr, &data) == S_ERROR)
		return (S_ERROR);

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

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

	/*
	 * The size of the hash table is determined by
	 *
	 *	i.	the initial nbucket and nchain entries (2)
	 *	ii.	the number of buckets (calculated above)
	 *	iii.	the number of chains (this is based on the number of
	 *		symbols in the .dynsym array).
	 */
	cnt = 2 + ofl->ofl_hashbkts + DYNSYM_ALL_CNT(ofl);
	size = cnt * shdr->sh_entsize;

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

	return (1);
}

/*
 * Generate the standard symbol table.  Contains all locals and globals,
 * and resides in a non-allocatable section (ie. it can be stripped).
 */
static uintptr_t
make_symtab(Ofl_desc *ofl)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;
	Is_desc		*xisec = 0;
	size_t		size;
	Word		symcnt;

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

	/*
	 * Place the section first since it will affect the local symbol
	 * count.
```

```
	 */
	if ((ofl->ofl_ossymtab = ld_place_section(ofl, isec, NULL,
	    ld_targ.t_id.id_symtab, NULL)) == (Os_desc *)S_ERROR)
		return (S_ERROR);

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

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

		if ((ofl->ofl_ossymshndx = ld_place_section(ofl, xisec, NULL,
		    ld_targ.t_id.id_symtab_ndx, NULL)) == (Os_desc *)S_ERROR)
			return (S_ERROR);
	}

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

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

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

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

	return (1);
}

/*
 * Build a dynamic symbol table. These tables reside in the text
 * segment of a dynamic executable or shared library.
 *
 *	.SUNW_ldynsym contains local function symbols
 *	.dynsym contains only globals symbols
 *
 * The two tables are created adjacent to each other, with .SUNW_ldynsym
 * coming first.
 */
static uintptr_t
make_dynsym(Ofl_desc *ofl)
{
	Shdr		*shdr, *lshdr;
	Elf_Data	*data, *ldata;
	Is_desc		*isec, *lisec;
	size_t		size;
	Xword		cnt;
	int		allow_ldynsym;

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

	/*
	 * Create the section headers. Note that we supply an ent_cnt
	 * of 0. We won't know the count until the section has been placed.
	 */
	if (allow_ldynsym && new_section(ofl, SHT_SUNW_LDYNSYM,
	    MSG_ORIG(MSG_SCN_LDYNSYM), 0, &lisec, &lshdr, &ldata) == S_ERROR)
		return (S_ERROR);

	if (new_section(ofl, SHT_DYNSYM, MSG_ORIG(MSG_SCN_DYNSYM), 0,
	    &isec, &shdr, &data) == S_ERROR)
		return (S_ERROR);

	/*
	 * Place the section(s) first since it will affect the local symbol
	 * count.
	 */
	if (allow_ldynsym &&
	    ((ofl->ofl_osldynsym = ld_place_section(ofl, lisec, NULL,
	    ld_targ.t_id.id_ldynsym, NULL)) == (Os_desc *)S_ERROR))
		return (S_ERROR);
	ofl->ofl_osdynsym =
	    ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_dynsym, NULL);
	if (ofl->ofl_osdynsym == (Os_desc *)S_ERROR)
		return (S_ERROR);

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

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

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

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

```
	}

	return (1);
}

/*
 * Build .SUNW_dynsymsort and/or .SUNW_dyntlssort sections. These are
 * index sections for the .SUNW_ldynsym/.dynsym pair that present data
 * and function symbols sorted by address.
 */
static uintptr_t
make_dynsort(Ofl_desc *ofl)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;

	/* Only do it if the .SUNW_ldynsym section is present */
	if (!OFL_ALLOW_LDYNSYM(ofl))
		return (1);

	/* .SUNW_dynsymsort */
	if (ofl->ofl_dynsymsortcnt > 0) {
		if (new_section(ofl, SHT_SUNW_symsort,
		    MSG_ORIG(MSG_SCN_DYNSYMSORT), ofl->ofl_dynsymsortcnt,
		    &isec, &shdr, &data) == S_ERROR)
		return (S_ERROR);

		if ((ofl->ofl_osdynsymsort = ld_place_section(ofl, isec, NULL,
		    ld_targ.t_id.id_dynsort, NULL)) == (Os_desc *)S_ERROR)
			return (S_ERROR);
	}

	/* .SUNW_dyntlssort */
	if (ofl->ofl_dyntlssortcnt > 0) {
		if (new_section(ofl, SHT_SUNW_tlssort,
		    MSG_ORIG(MSG_SCN_DYNTLSSORT),
		    ofl->ofl_dyntlssortcnt, &isec, &shdr, &data) == S_ERROR)
		return (S_ERROR);

		if ((ofl->ofl_osdyntlssort = ld_place_section(ofl, isec, NULL,
		    ld_targ.t_id.id_dynsort, NULL)) == (Os_desc *)S_ERROR)
			return (S_ERROR);
	}

	return (1);
}

/*
 * Helper routine for make_dynsym_shndx. Builds a
 * a SHT_SYMTAB_SHNDX for .dynsym or .SUNW_ldynsym, without knowing
 * which one it is.
 */
static uintptr_t
make_dyn_shndx(Ofl_desc *ofl, const char *shname, Os_desc *symtab,
    Os_desc **ret_os)
{
	Is_desc		*isec;
	Is_desc		*dynsymisp;
	Shdr		*shdr, *dynshdr;
	Elf_Data	*data;

	dynsymisp = ld_os_first_isdesc(symtab);
	dynshdr = dynsymisp->is_shdr;

	if (new_section(ofl, SHT_SYMTAB_SHNDX, shname,
            (dynshdr->sh_size / dynshdr->sh_entsize),
            &isec, &shdr, &data) == S_ERROR)
                return (S_ERROR);

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

        assert(*ret_os);

        return (1);
}

/*
 * Build a SHT_SYMTAB_SHNDX for the .dynsym, and .SUNW_ldynsym
 */
static uintptr_t
make_dynsym_shndx(Ofl_desc *ofl)
{
        /*
         * If there is a .SUNW_ldynsym, generate a section for its extended
         * index section as well.
         */
        if (OFL_ALLOW_LDYNSYM(ofl)) {
                if (make_dyn_shndx(ofl, MSG_ORIG(MSG_SCN_LDYNSYM_SHNDX),
                    ofl->ofl_osldynsym, &ofl->ofl_osldynshndx) == S_ERROR)
                        return (S_ERROR);
        }

        /* The Generate a section for the dynsym */
        if (make_dyn_shndx(ofl, MSG_ORIG(MSG_SCN_DYNSYM_SHNDX),
            ofl->ofl_osdynsym, &ofl->ofl_osdynshndx) == S_ERROR)
                return (S_ERROR);

        return (1);
}


/*
 * Build a string table for the section headers.
 */
static uintptr_t
make_shstrtab(Ofl_desc *ofl)
{
        Shdr            *shdr;
        Elf_Data        *data;
        Is_desc         *isec;
        size_t          size;

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

        /*
         * Place the section first, as it may effect the number of section
         * headers to account for.
         */
        ofl->ofl_osshstrtab =
            ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_note, NULL);
        if (ofl->ofl_osshstrtab == (Os_desc *)S_ERROR)
                return (S_ERROR);

        size = st_getstrtab_sz(ofl->ofl_shdrsttab);
        assert(size > 0);

        data->d_size = size;
```

```
        shdr->sh_size = (Xword)size;

        return (1);
}

/*
 * Build a string section for the standard symbol table.
 */
static uintptr_t
make_strtab(Ofl_desc *ofl)
{
        Shdr            *shdr;
        Elf_Data        *data;
        Is_desc         *isec;
        size_t          size;

        /*
         * This string table consists of all the global and local symbols.
         * Account for null bytes at end of the file name and the beginning
         * of section.
         */
        if (st_insert(ofl->ofl_strtab, ofl->ofl_name) == -1)
                return (S_ERROR);

        size = st_getstrtab_sz(ofl->ofl_strtab);
        assert(size > 0);

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

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

        ofl->ofl_osstrtab =
            ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_strtab, NULL);
        return ((uintptr_t)ofl->ofl_osstrtab);
}

/*
 * Build a string table for the dynamic symbol table.
 */
static uintptr_t
make_dynstr(Ofl_desc *ofl)
{
        Shdr            *shdr;
        Elf_Data        *data;
        Is_desc         *isec;
        size_t          size;

        /*
         * If producing a .SUNW_ldynsym, account for the initial STT_FILE
         * symbol that precedes the scope reduced global symbols.
         */
        if (OFL_ALLOW_LDYNSYM(ofl)) {
                if (st_insert(ofl->ofl_dynstrtab, ofl->ofl_name) == -1)
                        return (S_ERROR);
                ofl->ofl_dynscopecnt++;
        }

        /*
         * Account for any local, named register symbols.  These locals are
         * required for reference from DT_REGISTER .dynamic entries.
         */
        if (ofl->ofl_regsyms) {
```

```
		int	ndx;

		for (ndx = 0; ndx < ofl->ofl_regsymsno; ndx++) {
			Sym_desc	*sdp;

			if ((sdp = ofl->ofl_regsyms[ndx]) == NULL)
				continue;

			if (!SYM_IS_HIDDEN(sdp) &&
			    (ELF_ST_BIND(sdp->sd_sym->st_info) != STB_LOCAL))
				continue;

			if (sdp->sd_sym->st_name == NULL)
				continue;

			if (st_insert(ofl->ofl_dynstrtab, sdp->sd_name) == -1)
				return (S_ERROR);
		}
	}

	/*
	 * Reserve entries for any per-symbol auxiliary/filter strings.
	 */
	if (ofl->ofl_dtsfltrs != NULL) {
		Dfltr_desc	*dftp;
		Aliste		idx;

		for (ALIST_TRAVERSE(ofl->ofl_dtsfltrs, idx, dftp))
			if (st_insert(ofl->ofl_dynstrtab, dftp->dft_str) == -1)
				return (S_ERROR);
	}

	size = st_getstrtab_sz(ofl->ofl_dynstrtab);
	assert(size > 0);

	if (new_section(ofl, SHT_STRTAB, MSG_ORIG(MSG_SCN_DYNSTR),
	    0, &isec, &shdr, &data) == S_ERROR)
		return (S_ERROR);

	/* Make it allocable if necessary */
	if (!(ofl->ofl_flags & FLG_OF_RELOBJ))
		shdr->sh_flags |= SHF_ALLOC;

	/* Set the size of the data area */
	data->d_size = size + DYNSTR_EXTRA_PAD;

	shdr->sh_size = (Xword)size;

	ofl->ofl_osdynstr =
	    ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_dynstr, NULL);
	return ((uintptr_t)ofl->ofl_osdynstr);
}

/*
 * Generate an output relocation section which will contain the relocation
 * information to be applied to the `osp' section.
 *
 * If (osp == NULL) then we are creating the coalesced relocation section
 * for an executable and/or a shared object.
 */
static uintptr_t
make_reloc(Ofl_desc *ofl, Os_desc *osp)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;
```

```
	size_t		size;
	Xword		sh_flags;
	char		*sectname;
	Os_desc		*rosp;
	Word		relsize;
	const char	*rel_prefix;

	/* LINTED */
	if (ld_targ.t_m.m_rel_sht_type == SHT_REL) {
		/* REL */
		relsize = sizeof (Rel);
		rel_prefix = MSG_ORIG(MSG_SCN_REL);
	} else {
		/* RELA */
		relsize = sizeof (Rela);
		rel_prefix = MSG_ORIG(MSG_SCN_RELA);
	}

	if (osp) {
		size = osp->os_szoutrels;
		sh_flags = osp->os_shdr->sh_flags;
		if ((sectname = libld_malloc(strlen(rel_prefix) +
		    strlen(osp->os_name) + 1)) == 0)
			return (S_ERROR);
		(void) strcpy(sectname, rel_prefix);
		(void) strcat(sectname, osp->os_name);
	} else if (ofl->ofl_flags & FLG_OF_COMREL) {
		size = (ofl->ofl_reloccnt - ofl->ofl_reloccntsub) * relsize;
		sh_flags = SHF_ALLOC;
		sectname = (char *)MSG_ORIG(MSG_SCN_SUNWRELOC);
	} else {
		size = ofl->ofl_relocrelsz;
		sh_flags = SHF_ALLOC;
		sectname = (char *)rel_prefix;
	}

	/*
	 * Keep track of total size of 'output relocations' (to be stored
	 * in .dynamic)
	 */
	/* LINTED */
	ofl->ofl_relocsz += (Xword)size;

	if (new_section(ofl, ld_targ.t_m.m_rel_sht_type, sectname, 0, &isec,
	    &shdr, &data) == S_ERROR)
		return (S_ERROR);

	data->d_size = size;

	shdr->sh_size = (Xword)size;
	if (OFL_ALLOW_DYNSYM(ofl) && (sh_flags & SHF_ALLOC))
		shdr->sh_flags = SHF_ALLOC;

	if (osp) {
		/*
		 * The sh_info field of the SHT_REL* sections points to the
		 * section the relocations are to be applied to.
		 */
		shdr->sh_flags |= SHF_INFO_LINK;
	}

	rosp = ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_rel, NULL);
	if (rosp == (Os_desc *)S_ERROR)
		return (S_ERROR);

	/*
```

```
	 * Associate this relocation section to the section its going to
	 * relocate.
	 */
	if (osp) {
		Aliste	idx;
		Is_desc	*risp;

		/*
		 * This is used primarily so that we can update
		 * SHT_GROUP[sect_no] entries to point to the
		 * created output relocation sections.
		 */
		for (APLIST_TRAVERSE(osp->os_relisdescs, idx, risp)) {
			risp->is_osdesc = rosp;

			/*
			 * If the input relocation section had the SHF_GROUP
			 * flag set - propagate it to the output relocation
			 * section.
			 */
			if (risp->is_shdr->sh_flags & SHF_GROUP) {
				rosp->os_shdr->sh_flags |= SHF_GROUP;
				break;
			}
		}
		osp->os_relosdesc = rosp;
	} else
		ofl->ofl_osrel = rosp;

	/*
	 * If this is the first relocation section we've encountered save it
	 * so that the .dynamic entry can be initialized accordingly.
	 */
	if (ofl->ofl_osrelhead == (Os_desc *)0)
		ofl->ofl_osrelhead = rosp;

	return (1);
}

/*
 * Generate version needed section.
 */
static uintptr_t
make_verneed(Ofl_desc *ofl)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;

	/*
	 * verneed sections do not have a constant element size, so the
	 * value of ent_cnt specified here (0) is meaningless.
	 */
	if (new_section(ofl, SHT_SUNW_verneed, MSG_ORIG(MSG_SCN_SUNWVERSION),
	    0, &isec, &shdr, &data) == S_ERROR)
		return (S_ERROR);

	/* During version processing we calculated the total size. */
	data->d_size = ofl->ofl_verneedsz;
	shdr->sh_size = (Xword)ofl->ofl_verneedsz;

	ofl->ofl_osverneed =
	    ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_version, NULL);
	return ((uintptr_t)ofl->ofl_osverneed);
}
```

```
/*
 * Generate a version definition section.
 *
 *  o	the SHT_SUNW_verdef section defines the versions that exist within this
 *	image.
 */
static uintptr_t
make_verdef(Ofl_desc *ofl)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;
	Ver_desc	*vdp;
	Str_tbl		*strtab;

	/*
	 * Reserve a string table entry for the base version dependency (other
	 * dependencies have symbol representations, which will already be
	 * accounted for during symbol processing).
	 */
	vdp = (Ver_desc *)ofl->ofl_verdesc->apl_data[0];

	if (OFL_IS_STATIC_OBJ(ofl))
		strtab = ofl->ofl_strtab;
	else
		strtab = ofl->ofl_dynstrtab;

	if (st_insert(strtab, vdp->vd_name) == -1)
		return (S_ERROR);

	/*
	 * verdef sections do not have a constant element size, so the
	 * value of ent_cnt specified here (0) is meaningless.
	 */
	if (new_section(ofl, SHT_SUNW_verdef, MSG_ORIG(MSG_SCN_SUNWVERSION),
	    0, &isec, &shdr, &data) == S_ERROR)
		return (S_ERROR);

	/* During version processing we calculated the total size. */
	data->d_size = ofl->ofl_verdefsz;
	shdr->sh_size = (Xword)ofl->ofl_verdefsz;

	ofl->ofl_osverdef =
	    ld_place_section(ofl, isec, NULL, ld_targ.t_id.id_version, NULL);
	return ((uintptr_t)ofl->ofl_osverdef);
}

/*
 * This routine is called when -z nopartial is in effect.
 */
uintptr_t
ld_make_parexpn_data(Ofl_desc *ofl, size_t size, Xword align)
{
	Shdr		*shdr;
	Elf_Data	*data;
	Is_desc		*isec;
	Os_desc		*osp;

	if (new_section(ofl, SHT_PROGBITS, MSG_ORIG(MSG_SCN_DATA), 0,
	    &isec, &shdr, &data) == S_ERROR)
		return (S_ERROR);

	shdr->sh_flags |= SHF_WRITE;
	data->d_size = size;
	shdr->sh_size = (Xword)size;
	if (align != 0) {
```

```
                data->d_align = align;
                shdr->sh_addralign = align;
        }

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

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

        if (!(osp->os_flags & FLG_OS_OUTREL)) {
                ofl->ofl_dynshdrcnt++;
                osp->os_flags |= FLG_OS_OUTREL;
        }
        return (1);
}

/*
 * Make .sunwmove section
 */
uintptr_t
ld_make_sunwmove(Ofl_desc *ofl, int mv_nums)
{
        Shdr            *shdr;
        Elf_Data        *data;
        Is_desc         *isec;
        Aliste          idx;
        Sym_desc        *sdp;
        int             cnt = 1;


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

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

        /*
         * Copy move entries
         */
        for (APLIST_TRAVERSE(ofl->ofl_parsyms, idx, sdp)) {
                Aliste          idx2;
                Mv_desc         *mdp;

                if (sdp->sd_flags & FLG_SY_PAREXPN)
                        continue;

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

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

        return (1);
}
```

```
/*
 * Given a relocation descriptor that references a string table
 * input section, locate the string referenced and return a pointer
 * to it.
 */
static const char *
strmerge_get_reloc_str(Ofl_desc *ofl, Rel_desc *rsp)
{
        Sym_desc *sdp = rsp->rel_sym;
        Xword    str_off;

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

        if (ELF_ST_TYPE(sdp->sd_sym->st_info) != STT_SECTION) {
                str_off = sdp->sd_sym->st_value;
        } else if ((rsp->rel_flags & FLG_REL_RELA) == FLG_REL_RELA) {
                /*
                 * For SHT_RELA, the addend value is found in the
                 * rel_raddend field of the relocation.
                 */
                str_off = rsp->rel_raddend;
        } else {        /* REL and STT_SECTION */
                /*
                 * For SHT_REL, the "addend" is not part of the relocation
                 * record. Instead, it is found at the relocation target
                 * address.
                 */
                uchar_t *addr = (uchar_t *)((uintptr_t)rsp->rel_roffset +
                    (uintptr_t)rsp->rel_isdesc->is_indata->d_buf);

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

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

/*
 * First pass over the relocation records for string table merging.
 * Build lists of relocations and symbols that will need modification,
 * and insert the strings they reference into the mstrtab string table.
 *
 * entry:
 *      ofl, osp - As passed to ld_make_strmerge().
 *      mstrtab - String table to receive input strings. This table
 *              must be in its first (initialization) pass and not
 *              yet cooked (st_getstrtab_sz() not yet called).
 *      rel_alpp - APlist to receive pointer to any relocation
 *              descriptors with STT_SECTION symbols that reference
 *              one of the input sections being merged.
 *      sym_alpp - APlist to receive pointer to any symbols that reference
 *              one of the input sections being merged.
 *      rcp - Pointer to cache of relocation descriptors to examine.
 *              Either &ofl->ofl_actrels (active relocations)
 *              or &ofl->ofl_outrels (output relocations).
 *
 * exit:
 *      On success, rel_alpp and sym_alpp are updated, and
 *      any strings in the mergeable input sections referenced by
 *      a relocation has been entered into mstrtab. True (1) is returned.
 *
 *      On failure, False (0) is returned.
 */
static int
strmerge_pass1(Ofl_desc *ofl, Os_desc *osp, Str_tbl *mstrtab,
    APlist **rel_alpp, APlist **sym_alpp, Rel_cache *rcp)
{
        Aliste          idx;
        Rel_cachebuf    *rcbp;
        Sym_desc        *sdp;
        Sym_desc        *last_sdp = NULL;
        Rel_desc        *rsp;
        const char      *name;

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

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

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

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

        return (1);
}

/*
 * If the output section has any SHF_MERGE|SHF_STRINGS input sections,
 * replace them with a single merged/compressed input section.
 *
 * entry:
 *      ofl - Output file descriptor
 *      osp - Output section descriptor
 *      rel_alpp, sym_alpp, - Address of 2 APlists, to be used
 *              for internal processing. On the initial call to
 *              ld_make_strmerge, these list pointers must be NULL.
 *              The caller is encouraged to pass the same lists back for
 *              successive calls to this function without freeing
 *              them in between calls. This causes a single pair of
 *              memory allocations to be reused multiple times.
 *
 * exit:
```

```
 *      If section merging is possible, it is done. If no errors are
 *      encountered, True (1) is returned. On error, S_ERROR.
 *
 *      The contents of rel_alpp and sym_alpp on exit are
 *      undefined. The caller can free them, or pass them back to a subsequent
 *      call to this routine, but should not examine their contents.
 */
static uintptr_t
ld_make_strmerge(Ofl_desc *ofl, Os_desc *osp, APlist **rel_alpp,
    APlist **sym_alpp)
{
        Str_tbl         *mstrtab;       /* string table for string merge secs */
        Is_desc         *mstrsec;       /* Generated string merge section */
        Is_desc         *isp;
        Shdr            *mstr_shdr;
        Elf_Data        *mstr_data;
        Sym_desc        *sdp;
        Rel_desc        *rsp;
        Aliste          idx;
        size_t          data_size;
        int             st_setstring_status;
        size_t          stoff;

        /* If string table compression is disabled, there's nothing to do */
        if ((ofl->ofl_flags1 & FLG_OF1_NCSTTAB) != 0)
                return (1);

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

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

                /*
                 * We have at least one non-discarded section.
                 * Create a string table descriptor.
                 */
                if ((mstrtab = st_new(FLG_STNEW_COMPRESS)) == NULL)
                        return (S_ERROR);
                break;
        }

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

        /*
         * This routine has to make 3 passes:
         *
         *      1) Examine all relocations, insert strings from relocations
         *              to the mergeable input sections into the string table.
         *      2) Modify the relocation values to be correct for the
         *              new merged section.
         *      3) Modify the symbols used by the relocations to reference
         *              the new section.
         *
```

```
	 * These passes cannot be combined:
	 *	- The string table code works in two passes, and all
	 *		strings have to be loaded in pass one before the
	 *		offset of any strings can be determined.
	 *	- Multiple relocations reference a single symbol, so the
	 *		symbol cannot be modified until all relocations are
	 *		fixed.
	 *
	 * The number of relocations related to section merging is usually
	 * a mere fraction of the overall active and output relocation lists,
	 * and the number of symbols is usually a fraction of the number
	 * of related relocations. We therefore build APlists for the
	 * relocations and symbols in the first pass, and then use those
	 * lists to accelerate the operation of pass 2 and 3.
	 *
	 * Reinitialize the lists to a completely empty state.
	 */
	aplist_reset(*rel_alpp);
	aplist_reset(*sym_alpp);

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

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

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

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

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

	/*
```

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

		/* Put the string into the merged string table */
		name = strmerge_get_reloc_str(ofl, rsp);
		st_setstring_status = st_setstring(mstrtab, name, &stoff);
		if (st_setstring_status == -1) {
			/*
			 * A failure to insert at this point means that
			 * something is corrupt. This isn’t a resource issue.
			 */
			assert(st_setstring_status != -1);
			goto return_s_error;
		}

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

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

	/*
	 * Pass 3:
	 *
	 * Modify the symbols referenced by the relocation descriptors
	 * so that they reference the new input section containing the
	 * merged strings instead of the original input sections.
	 */
	for (APLIST_TRAVERSE(*sym_alpp, idx, sdp)) {
		/*
		 * If we’ve already processed this symbol, don’t do it
		 * twice. strmerge_pass1() uses a heuristic (relocations to
		 * the same symbol clump together) to avoid inserting a
		 * given symbol more than once, but repeat symbols in
		 * the list can occur.
		 */
		if ((sdp->sd_isc->is_flags & FLG_IS_INSTRMRG) == 0)
			continue;

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

			st_setstring_status =
			    st_setstring(mstrtab, name, &stoff);
			if (st_setstring_status == -1) {
				/*
				 * A failure to insert at this point means
				 * something is corrupt. This isn't a
				 * resource issue.
				 */
				assert(st_setstring_status != -1);
				goto return_s_error;
			}

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

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

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

		data_size += isp->is_indata->d_size;

		isp->is_flags |= FLG_IS_DISCARD;
		DBG_CALL(Dbg_sec_discarded(ofl->ofl_lml, isp, mstrsec));
	}

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

	st_destroy(mstrtab);
	return (1);

return_s_error:
	st_destroy(mstrtab);
	return (S_ERROR);
}

/*
```

```
 * Update a data buffers size.  A number of sections have to be created, and
 * the sections header contributes to the size of the eventual section.  Thus,
 * a section may be created, and once all associated sections have been created,
 * we return to establish the required section size.
 */
inline static void
update_data_size(Os_desc *osp, ulong_t cnt)
{
	Is_desc		*isec = ld_os_first_isdesc(osp);
	Elf_Data	*data = isec->is_indata;
	Shdr		*shdr = osp->os_shdr;
	size_t		size = cnt * shdr->sh_entsize;

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

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

	/*
	 * Generate any special sections.
	 */
	if (flags & FLG_OF_ADDVERS)
		if (make_comment(ofl) == S_ERROR)
			return (S_ERROR);

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

	/*
	 * Create a capabilities section if required.
	 */
	if (make_cap(ofl, SHT_SUNW_cap, MSG_ORIG(MSG_SCN_SUNWCAP),
	    ld_targ.t_id.id_cap) == S_ERROR)
		return (S_ERROR);

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

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

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

	/*
	 * Make the .plt section.  This occurs after any other relocation
	 * sections are generated (see reloc_init()) to ensure that the
	 * associated relocation section is after all the other relocation
	 * sections.
	 */
	if ((ofl->ofl_pltcnt) || (ofl->ofl_pltpad))
		if (make_plt(ofl) == S_ERROR)
			return (S_ERROR);

	/*
	 * Determine whether any sections or files are not referenced.  Under
	 * -Dunused a diagnostic for any unused components is generated, under
	 * -zignore the component is removed from the final output.
	 */
	if (DBG_ENABLED || (ofl->ofl_flags1 & FLG_OF1_IGNPRC)) {
		if (ignore_section_processing(ofl) == S_ERROR)
			return (S_ERROR);
	}

	/*
	 * If we have detected a situation in which previously placed
	 * output sections may have been discarded, perform the necessary
	 * readjustment.
	 */
	if (ofl->ofl_flags & FLG_OF_ADJOSCNT)
		adjust_os_count(ofl);

	/*
	 * Do any of the output sections contain input sections that
	 * are candidates for string table merging? For each such case,
	 * we create a replacement section, insert it, and discard the
	 * originals.
	 *
	 * rel_alpp and sym_alpp are used by ld_make_strmerge()
	 * for its internal processing. We are responsible for the
	 * initialization and cleanup, and ld_make_strmerge() handles the rest.
	 * This allows us to reuse a single pair of memory buffers, allocated
	 * for this processing, for all the output sections.
	 */
	if ((ofl->ofl_flags1 & FLG_OF1_NCSTTAB) == 0) {
		int	error_seen = 0;
		APlist	*rel_alpp = NULL;
		APlist	*sym_alpp = NULL;
		Aliste	idx1;

		for (APLIST_TRAVERSE(ofl->ofl_segs, idx1, sgp)) {
			Os_desc	*osp;
			Aliste	idx2;

			for (APLIST_TRAVERSE(sgp->sg_osdescs, idx2, osp))
				if ((osp->os_mstrisdescs != NULL) &&
				    (ld_make_strmerge(ofl, osp,
				    &rel_alpp, &sym_alpp) ==
				    S_ERROR)) {
					error_seen = 1;
					break;
				}
		}
		if (rel_alpp != NULL)
			libld_free(rel_alpp);
		if (sym_alpp != NULL)
			libld_free(sym_alpp);
		if (error_seen != 0)
			return (S_ERROR);
	}

	/*
	 * Add any necessary versioning information.
	 */
```

```
	if (!(flags & FLG_OF_NOVERSEC)) {
		if ((flags & FLG_OF_VERNEED) &&
		    (make_verneed(ofl) == S_ERROR))
			return (S_ERROR);
		if ((flags & FLG_OF_VERDEF) &&
		    (make_verdef(ofl) == S_ERROR))
			return (S_ERROR);
		if ((flags & (FLG_OF_VERNEED | FLG_OF_VERDEF)) &&
		    ((ofl->ofl_osversym = make_sym_sec(ofl,
		    MSG_ORIG(MSG_SCN_SUNWVERSYM), SHT_SUNW_versym,
		    ld_targ.t_id.id_version)) == (Os_desc*)S_ERROR))
			return (S_ERROR);
	}

	/*
	 * Create a syminfo section if necessary.
	 */
	if (flags & FLG_OF_SYMINFO) {
		if ((ofl->ofl_ossyminfo = make_sym_sec(ofl,
		    MSG_ORIG(MSG_SCN_SUNWSYMINFO), SHT_SUNW_syminfo,
		    ld_targ.t_id.id_syminfo)) == (Os_desc *)S_ERROR)
			return (S_ERROR);
	}

	if (flags & FLG_OF_COMREL) {
		/*
		 * If -zcombreloc is enabled then all relocations (except for
		 * the PLT’s) are coalesced into a single relocation section.
		 */
		if (ofl->ofl_reloccnt) {
			if (make_reloc(ofl, NULL) == S_ERROR)
				return (S_ERROR);
		}
	} else {
		Aliste	idx1;

		/*
		 * Create the required output relocation sections.  Note, new
		 * sections may be added to the section list that is being
		 * traversed.  These insertions can move the elements of the
		 * Alist such that a section descriptor is re-read.  Recursion
		 * is prevented by maintaining a previous section pointer and
		 * insuring that this pointer isn’t re-examined.
		 */
		for (APLIST_TRAVERSE(ofl->ofl_segs, idx1, sgp)) {
			Os_desc	*osp, *posp = 0;
			Aliste	idx2;

			for (APLIST_TRAVERSE(sgp->sg_osdescs, idx2, osp)) {
				if ((osp != posp) && osp->os_szoutrels &&
				    (osp != ofl->ofl_osplt)) {
					if (make_reloc(ofl, osp) == S_ERROR)
						return (S_ERROR);
				}
				posp = osp;
			}
		}

		/*
		 * If we’re not building a combined relocation section, then
		 * build a .rel[a] section as required.
		 */
		if (ofl->ofl_relocrelsz) {
			if (make_reloc(ofl, NULL) == S_ERROR)
				return (S_ERROR);
		}
```

```
        }

        /*
         * The PLT relocations are always in their own section, and we try to
         * keep them at the end of the PLT table.  We do this to keep the hot
         * "data" PLT's at the head of the table nearer the .dynsym & .hash.
         */
        if (ofl->ofl_osplt && ofl->ofl_relocpltsz) {
                if (make_reloc(ofl, ofl->ofl_osplt) == S_ERROR)
                        return (S_ERROR);
        }

        /*
         * Finally build the symbol and section header sections.
         */
        if (flags & FLG_OF_DYNAMIC) {
                if (make_dynamic(ofl) == S_ERROR)
                        return (S_ERROR);

                /*
                 * A number of sections aren't necessary within a relocatable
                 * object, even if -dy has been used.
                 */
                if (!(flags & FLG_OF_RELOBJ)) {
                        if (make_hash(ofl) == S_ERROR)
                                return (S_ERROR);
                        if (make_dynstr(ofl) == S_ERROR)
                                return (S_ERROR);
                        if (make_dynsym(ofl) == S_ERROR)
                                return (S_ERROR);
                        if (ld_unwind_make_hdr(ofl) == S_ERROR)
                                return (S_ERROR);
                        if (make_dynsort(ofl) == S_ERROR)
                                return (S_ERROR);
                }
        }

        if (!(flags & FLG_OF_STRIP) || (flags & FLG_OF_RELOBJ) ||
            ((flags & FLG_OF_STATIC) && ofl->ofl_osversym)) {
                /*
                 * Do we need to make a SHT_SYMTAB_SHNDX section
                 * for the dynsym.  If so - do it now.
                 */
                if (ofl->ofl_osdynsym &&
                    ((ofl->ofl_shdrcnt + 3) >= SHN_LORESERVE)) {
                        if (make_dynsym_shndx(ofl) == S_ERROR)
                                return (S_ERROR);
                }

                if (make_strtab(ofl) == S_ERROR)
                        return (S_ERROR);
                if (make_symtab(ofl) == S_ERROR)
                        return (S_ERROR);
        } else {
                /*
                 * Do we need to make a SHT_SYMTAB_SHNDX section
                 * for the dynsym.  If so - do it now.
                 */
                if (ofl->ofl_osdynsym &&
                    ((ofl->ofl_shdrcnt + 1) >= SHN_LORESERVE)) {
                        if (make_dynsym_shndx(ofl) == S_ERROR)
                                return (S_ERROR);
                }
        }

        if (make_shstrtab(ofl) == S_ERROR)
```

```
                return (S_ERROR);

        /*
         * Now that we've created all output sections, adjust the size of the
         * SHT_SUNW_versym and SHT_SUNW_syminfo section, which are dependent on
         * the associated symbol table sizes.
         */
        if (ofl->ofl_osversym || ofl->ofl_ossyminfo) {
                ulong_t         cnt;
                Is_desc         *isp;
                Os_desc         *osp;

                if (OFL_IS_STATIC_OBJ(ofl))
                        osp = ofl->ofl_ossymtab;
                else
                        osp = ofl->ofl_osdynsym;

                isp = ld_os_first_isdesc(osp);
                cnt = (isp->is_shdr->sh_size / isp->is_shdr->sh_entsize);

                if (ofl->ofl_osversym)
                        update_data_size(ofl->ofl_osversym, cnt);

                if (ofl->ofl_ossyminfo)
                        update_data_size(ofl->ofl_ossyminfo, cnt);
        }

        /*
         * Now that we've created all output sections, adjust the size of the
         * SHT_SUNW_capinfo, which is dependent on the associated symbol table
         * size.
         */
        if (ofl->ofl_oscapinfo) {
                ulong_t cnt;

                /*
                 * Symbol capabilities symbols are placed directly after the
                 * STT_FILE symbol, section symbols, and any register symbols.
                 * Effectively these are the first of any series of demoted
                 * (scoped) symbols.
                 */
                if (OFL_IS_STATIC_OBJ(ofl))
                        cnt = SYMTAB_ALL_CNT(ofl);
                else
                        cnt = DYNSYM_ALL_CNT(ofl);

                update_data_size(ofl->ofl_oscapinfo, cnt);
        }
        return (1);
}

/*
 * Build an additional data section - used to back OBJT symbol definitions
 * added with a mapfile.
 */
Is_desc *
ld_make_data(Ofl_desc *ofl, size_t size)
{
        Shdr            *shdr;
        Elf_Data        *data;
        Is_desc         *isec;

        if (new_section(ofl, SHT_PROGBITS, MSG_ORIG(MSG_SCN_DATA), 0,
            &isec, &shdr, &data) == S_ERROR)
                return ((Is_desc *)S_ERROR);
        data->d_size = size;
        shdr->sh_size = (Xword)size;
        shdr->sh_flags |= SHF_WRITE;

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

        return (isec);
}

/*
 * Build an additional text section - used to back FUNC symbol definitions
 * added with a mapfile.
 */
Is_desc *
ld_make_text(Ofl_desc *ofl, size_t size)
{
        Shdr            *shdr;
        Elf_Data        *data;
        Is_desc         *isec;

        /*
         * Insure the size is sufficient to contain the minimum return
         * instruction.
         */
        if (size < ld_targ.t_nf.nf_size)
                size = ld_targ.t_nf.nf_size;

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

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

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

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

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

        return (isec);
}

void
ld_comdat_validate(Ofl_desc *ofl, Ifl_desc *ifl)
{
        int i;
```

```
        for (i = 0; i < ifl->ifl_shnum; i++) {
                Is_desc *isp = ifl->ifl_isdesc[i];
                int types = 0;
                char buf[1024] = "";
                Group_desc *gr = NULL;

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

                if (isp->is_shdr->sh_type == SHT_SUNW_COMDAT) {
                        types++;
                        (void) strlcpy(buf, MSG_ORIG(MSG_STR_SUNW_COMDAT),
                            sizeof (buf));
                }

                if (strncmp(MSG_ORIG(MSG_SCN_GNU_LINKONCE), isp->is_name,
                    MSG_SCN_GNU_LINKONCE_SIZE) == 0) {
                        types++;
                        if (types > 1)
                                (void) strlcat(buf, ", ", sizeof (buf));
                        (void) strlcat(buf, MSG_ORIG(MSG_SCN_GNU_LINKONCE),
                            sizeof (buf));
                }

                if ((isp->is_shdr->sh_flags & SHF_GROUP) &&
                    ((gr = ld_get_group(ofl, isp)) != NULL) &&
                    (gr->gd_data[0] & GRP_COMDAT)) {
                        types++;
                        if (types > 1)
                                (void) strlcat(buf, ", ", sizeof (buf));
                        (void) strlcat(buf, MSG_ORIG(MSG_STR_GROUP),
                            sizeof (buf));
                }

                if (types > 1)
                        ld_eprintf(ofl, ERR_FATAL,
                            MSG_INTL(MSG_SCN_MULTICOMDAT), ifl->ifl_name,
                            EC_WORD(isp->is_scnndx), isp->is_name, buf);
        }
}
```

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

/*
 * At this point all symbol input processing has been completed, therefore
 * complete the symbol table entries by generating any necessary internal
 * symbols.
 */
uintptr_t
ld_sym_spec(Ofl_desc *ofl)
{
        Sym_desc        *sdp;
        Sg_desc         *sgp;
        Aliste          idx1;

        DBG_CALL(Dbg_syms_spec_title(ofl->ofl_lml));

        /*
         * For each section in the output file, look for symbols named for the
         * __start/__stop patterns.  If references exist, flesh the symbols to
         * be defined.
         *
         * The symbols are given values at the same time as the other special
         * symbols.
         */
        if (!(ofl->ofl_flags & FLG_OF_RELOBJ) ||
            (ofl->ofl_flags & FLG_OF_KMOD)) {
                Aliste          idx1;

#endif /* ! codereview */
                for (APLIST_TRAVERSE(ofl->ofl_segs, idx1, sgp)) {
                        Os_desc *osp;
                        Aliste idx2;

                        for (APLIST_TRAVERSE(sgp->sg_osdescs, idx2, osp)) {
                                if (is_cname(osp->os_name)) {
                                        sym_add_bounds(ofl, osp,
                                            SDAUX_ID_SECBOUND_START);
                                        sym_add_bounds(ofl, osp,
                                            SDAUX_ID_SECBOUND_STOP);
                                }
                        }
                }
        }

        if (ofl->ofl_flags & FLG_OF_RELOBJ)
                return (1);

        if (sym_add_spec(MSG_ORIG(MSG_SYM_ETEXT), MSG_ORIG(MSG_SYM_ETEXT_U),
            SDAUX_ID_ETEXT, 0, (FLG_SY_DEFAULT | FLG_SY_EXPDEF),
            ofl) == S_ERROR)
                return (S_ERROR);
        if (sym_add_spec(MSG_ORIG(MSG_SYM_EDATA), MSG_ORIG(MSG_SYM_EDATA_U),
            SDAUX_ID_EDATA, 0, (FLG_SY_DEFAULT | FLG_SY_EXPDEF),
            ofl) == S_ERROR)
                return (S_ERROR);
        if (sym_add_spec(MSG_ORIG(MSG_SYM_END), MSG_ORIG(MSG_SYM_END_U),
            SDAUX_ID_END, FLG_SY_DYNSORT, (FLG_SY_DEFAULT | FLG_SY_EXPDEF),
            ofl) == S_ERROR)
                return (S_ERROR);
        if (sym_add_spec(MSG_ORIG(MSG_SYM_L_END), MSG_ORIG(MSG_SYM_L_END_U),
            SDAUX_ID_END, 0, FLG_SY_HIDDEN, ofl) == S_ERROR)
```

```
                return (S_ERROR);
        if (sym_add_spec(MSG_ORIG(MSG_SYM_L_START), MSG_ORIG(MSG_SYM_L_START_U),
            SDAUX_ID_START, 0, FLG_SY_HIDDEN, ofl) == S_ERROR)
                return (S_ERROR);

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

        if (OFL_ALLOW_DYNSYM(ofl))
                if (sym_add_spec(MSG_ORIG(MSG_SYM_PLKTBL),
                    MSG_ORIG(MSG_SYM_PLKTBL_U), SDAUX_ID_PLT,
                    FLG_SY_DYNSORT, (FLG_SY_DEFAULT | FLG_SY_EXPDEF),
                    ofl) == S_ERROR)
                        return (S_ERROR);

        /*
         * A GOT reference will be accompanied by the associated GOT symbol.
         * Make sure it gets assigned the appropriate special attributes.
         */
        if (((sdp = ld_sym_find(MSG_ORIG(MSG_SYM_GOFTBL_U),
            SYM_NOHASH, NULL, ofl)) != NULL) && (sdp->sd_ref != REF_DYN_SEEN)) {
                if (sym_add_spec(MSG_ORIG(MSG_SYM_GOFTBL),
                    MSG_ORIG(MSG_SYM_GOFTBL_U), SDAUX_ID_GOT, FLG_SY_DYNSORT,
                    (FLG_SY_DEFAULT | FLG_SY_EXPDEF), ofl) == S_ERROR)
                        return (S_ERROR);
        }

        return (1);
}

/*
 * Determine a potential capability symbol's visibility.
 *
 * The -z symbolcap option transforms an object capabilities relocatable object
 * into a symbol capabilities relocatable object.  Any global function symbols,
 * or initialized global data symbols are candidates for transforming into local
 * symbol capabilities definitions.  However, if a user indicates that a symbol
 * should be demoted to local using a mapfile, then there is no need to
 * transform the associated global symbol.
 *
 * Normally, a symbol's visibility is determined after the symbol resolution
 * process, after all symbol state has been gathered and resolved.  However,
 * for -z symbolcap, this determination is too late.  When a global symbol is
 * read from an input file we need to determine it's visibility so as to decide
 * whether to create a local or not.
 *
 * If a user has explicitly defined this symbol as having local scope within a
 * mapfile, then a symbol of the same name already exists.  However, explicit
 * local definitions are uncommon, as most mapfiles define the global symbol
 * requirements together with an auto-reduction directive '*'.  If this state
 * has been defined, then we must make sure that the new symbol isn't a type
 * that can not be demoted to local.
 */
static int
sym_cap_vis(const char *name, Word hash, Sym *sym, Ofl_desc *ofl)
{
        Sym_desc        *sdp;
        uchar_t         vis;
        avl_index_t     where;
        sd_flag_t       sdflags = 0;
	/*
	 * Determine the visibility of the new symbol.
	 */
	vis = ELF_ST_VISIBILITY(sym->st_other);
	switch (vis) {
	case STV_EXPORTED:
		sdflags |= FLG_SY_EXPORT;
		break;
	case STV_SINGLETON:
		sdflags |= FLG_SY_SINGLE;
		break;
	}

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

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

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

/*
 * This routine checks to see if a symbols visibility needs to be reduced to
 * either SYMBOLIC or LOCAL.  This routine can be called from either
 * reloc_init() or sym_validate().
 */
void
ld_sym_adjust_vis(Sym_desc *sdp, Ofl_desc *ofl)
{
	ofl_flag_t	oflags = ofl->ofl_flags;
	Sym		*sym = sdp->sd_sym;

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

```
				    (FLG_SY_REDUCED | FLG_SY_HIDDEN);
			}

			if (oflags & (FLG_OF_REDLSYM | FLG_OF_AUTOELM)) {
				sdp->sd_flags |= FLG_SY_ELIM;
				sym->st_other = STV_ELIMINATE |
				    (sym->st_other & ~MSK_SYM_VISIBILITY);
			} else if (ELF_ST_VISIBILITY(sym->st_other) !=
			    STV_INTERNAL)
				sym->st_other = STV_HIDDEN |
				    (sym->st_other & ~MSK_SYM_VISIBILITY);
		}

		/*
		 * If -Bsymbolic is in effect, and the symbol hasn't explicitly
		 * been defined nodirect (via a mapfile), then bind the global
		 * symbol symbolically and assign the STV_PROTECTED visibility
		 * attribute.
		 */
		if ((oflags & FLG_OF_SYMBOLIC) &&
		    ((sdp->sd_flags & (FLG_SY_HIDDEN | FLG_SY_NDIR)) == 0)) {
			sdp->sd_flags |= FLG_SY_PROTECT;
			if (ELF_ST_VISIBILITY(sym->st_other) == STV_DEFAULT)
				sym->st_other = STV_PROTECTED |
				    (sym->st_other & ~MSK_SYM_VISIBILITY);
		}
	}

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

/*
 * Make sure a symbol definition is local to the object being built.
 */
inline static int
ensure_sym_local(Ofl_desc *ofl, Sym_desc *sdp, const char *str)
{
	if (sdp->sd_sym->st_shndx == SHN_UNDEF) {
		if (str) {
			ld_eprintf(ofl, ERR_FATAL, MSG_INTL(MSG_SYM_UNDEF),
			    str, demangle((char *)sdp->sd_name));
		}
		return (1);
	}
	if (sdp->sd_ref != REF_REL_NEED) {
		if (str) {
			ld_eprintf(ofl, ERR_FATAL, MSG_INTL(MSG_SYM_EXTERN),
			    str, demangle((char *)sdp->sd_name),
			    sdp->sd_file->ifl_name);
		}
		return (1);
	}

	sdp->sd_flags |= FLG_SY_UPREQD;
	if (sdp->sd_isc) {
		sdp->sd_isc->is_flags |= FLG_IS_SECTREF;
		sdp->sd_isc->is_file->ifl_flags |= FLG_IF_FILEREF;
	}
	return (0);
}

/*
```

```
 * Make sure all the symbol definitions required for initarray, finiarray, or
 * preinitarray's are local to the object being built.
 */
static int
ensure_array_local(Ofl_desc *ofl, APlist *apl, const char *str)
{
        Aliste          idx;
        Sym_desc        *sdp;
        int             ret = 0;

        for (APLIST_TRAVERSE(apl, idx, sdp))
                ret += ensure_sym_local(ofl, sdp, str);

        return (ret);
}

/*
 * After all symbol table input processing has been finished, and all relocation
 * counting has been carried out (ie. no more symbols will be read, generated,
 * or modified), validate and count the relevant entries:
 *
 *  -   check and print any undefined symbols remaining.  Note that if a symbol
 *      has been defined by virtue of the inclusion of  an implicit shared
 *      library, it is still classed as undefined.
 *
 *  -   count the number of global needed symbols together with the size of
 *      their associated name strings (if scoping has been indicated these
 *      symbols may be reduced to locals).
 *
 *  -   establish the size and alignment requirements for the global .bss
 *      section (the alignment of this section is based on the first symbol
 *      that it will contain).
 */
uintptr_t
ld_sym_validate(Ofl_desc *ofl)
{
        Sym_avlnode     *sav;
        Sym_desc        *sdp;
        Sym             *sym;
        ofl_flag_t      oflags = ofl->ofl_flags;
        ofl_flag_t      undef = 0, needed = 0, verdesc = 0;
        Xword           bssalign = 0, tlsalign = 0;
        Boolean         need_bss, need_tlsbss;
        Xword           bsssize = 0, tlssize = 0;
#if     defined(_ELF64)
        Xword           lbssalign = 0, lbsssize = 0;
        Boolean         need_lbss;
#endif
        int             ret, allow_ldynsym;
        uchar_t         type;
        ofl_flag_t      undef_state = 0;

        DBG_CALL(Dbg_basic_validate(ofl->ofl_lml));

        /*
         * The need_XXX booleans are used to determine whether we need to
         * create each type of bss section. We used to create these sections
         * if the sum of the required sizes for each type were non-zero.
         * However, it is possible for a compiler to generate COMMON variables
         * of zero-length and this tricks that logic --- even zero-length
         * symbols need an output section.
         */
        need_bss = need_tlsbss = FALSE;
#if     defined(_ELF64)
        need_lbss = FALSE;
#endif
```

```
        /*
         * Determine how undefined symbols are handled:
         *
         * fatal:
         *      If this link-edit calls for no undefined symbols to remain
         *      (this is the default case when generating an executable but
         *      can be enforced for any object using -z defs), a fatal error
         *      condition will be indicated.
         *
         * warning:
         *      If we're creating a shared object, and either the -Bsymbolic
         *      flag is set, or the user has turned on the -z guidance feature,
         *      then a non-fatal warning is issued for each symbol.
         *
         * ignore:
         *      In all other cases, undefined symbols are quietly allowed.
         */
        if (oflags & FLG_OF_NOUNDEF) {
                undef = FLG_OF_FATAL;
        } else if (oflags & FLG_OF_SHAROBJ) {
                if ((oflags & FLG_OF_SYMBOLIC) ||
                    OFL_GUIDANCE(ofl, FLG_OFG_NO_DEFS))
                        undef = FLG_OF_WARN;
        }

        /*
         * If the symbol is referenced from an implicitly included shared object
         * (ie. it's not on the NEEDED list) then the symbol is also classified
         * as undefined and a fatal error condition will be indicated.
         */
        if ((oflags & FLG_OF_NOUNDEF) || !(oflags & FLG_OF_SHAROBJ))
                needed = FLG_OF_FATAL;
        else if ((oflags & FLG_OF_SHAROBJ) &&
            OFL_GUIDANCE(ofl, FLG_OFG_NO_DEFS))
                needed = FLG_OF_WARN;

        /*
         * If the output image is being versioned, then all symbol definitions
         * must be associated with a version.  Any symbol that isn't associated
         * with a version is classified as undefined, and a fatal error
         * condition is indicated.
         */
        if ((oflags & FLG_OF_VERDEF) && (ofl->ofl_vercnt > VER_NDX_GLOBAL))
                verdesc = FLG_OF_FATAL;

        allow_ldynsym = OFL_ALLOW_LDYNSYM(ofl);

        if (allow_ldynsym) {
                /*
                 * Normally, we disallow symbols with 0 size from appearing
                 * in a dyn[sym|tls]sort section. However, there are some
                 * symbols that serve special purposes that we want to exempt
                 * from this rule. Look them up, and set their
                 * FLG_SY_DYNSORT flag.
                 */
                static const char *special[] = {
                        MSG_ORIG(MSG_SYM_INIT_U),       /* _init */
                        MSG_ORIG(MSG_SYM_FINI_U),       /* _fini */
                        MSG_ORIG(MSG_SYM_START),        /* _start */
                        NULL
                };
                int i;

                for (i = 0; special[i] != NULL; i++) {
                        if (((sdp = ld_sym_find(special[i],
                            SYM_NOHASH, NULL, ofl)) != NULL) &&
                            (sdp->sd_sym->st_size == 0)) {
                                if (ld_sym_copy(sdp) == S_ERROR)
                                        return (S_ERROR);
                                sdp->sd_flags |= FLG_SY_DYNSORT;
                        }
                }
        }

        /*
         * Collect and validate the globals from the internal symbol table.
         */
        for (sav = avl_first(&ofl->ofl_symavl); sav;
            sav = AVL_NEXT(&ofl->ofl_symavl, sav)) {
                Is_desc         *isp;
                int             undeferr = 0;
                uchar_t         vis;

                sdp = sav->sav_sdp;

                /*
                 * If undefined symbols are allowed, and we're not being
                 * asked to supply guidance, ignore any symbols that are
                 * not needed.
                 */
                if (!(oflags & FLG_OF_NOUNDEF) &&
                    !OFL_GUIDANCE(ofl, FLG_OFG_NO_DEFS) &&
                    (sdp->sd_ref == REF_DYN_SEEN))
                        continue;

                /*
                 * If the symbol originates from an external or parent mapfile
                 * reference and hasn't been matched to a reference from a
                 * relocatable object, ignore it.
                 */
                if ((sdp->sd_flags & (FLG_SY_EXTERN | FLG_SY_PARENT)) &&
                    ((sdp->sd_flags & FLG_SY_MAPUSED) == 0)) {
                        sdp->sd_flags |= FLG_SY_INVALID;
                        continue;
                }

                sym = sdp->sd_sym;
                type = ELF_ST_TYPE(sym->st_info);

                /*
                 * Sanity check TLS.
                 */
                if ((type == STT_TLS) && (sym->st_size != 0) &&
                    (sym->st_shndx != SHN_UNDEF) &&
                    (sym->st_shndx != SHN_COMMON)) {
                        Is_desc         *isp = sdp->sd_isc;
                        Ifl_desc        *ifl = sdp->sd_file;

                        if ((isp == NULL) || (isp->is_shdr == NULL) ||
                            ((isp->is_shdr->sh_flags & SHF_TLS) == 0)) {
                                ld_eprintf(ofl, ERR_FATAL,
                                    MSG_INTL(MSG_SYM_TLS),
                                    demangle(sdp->sd_name), ifl->ifl_name);
                                continue;
                        }
                }

                if ((sdp->sd_flags & FLG_SY_VISIBLE) == 0)
                        ld_sym_adjust_vis(sdp, ofl);

                if ((sdp->sd_flags & FLG_SY_REDUCED) &&
```

```
                    (oflags & FLG_OF_PROCRED)) {
                        DBG_CALL(Dbg_syms_reduce(ofl, DBG_SYM_REDUCE_GLOBAL,
                            sdp, 0, 0));
                }

                /*
                 * Record any STV_SINGLETON existence.
                 */
                if ((vis = ELF_ST_VISIBILITY(sym->st_other)) == STV_SINGLETON)
                        ofl->ofl_dtflags_1 |= DF_1_SINGLETON;

                /*
                 * If building a shared object or executable, and this is a
                 * non-weak UNDEF symbol with reduced visibility (STV_*), then
                 * give a fatal error.
                 */
                if (((oflags & FLG_OF_RELOBJ) == 0) &&
                    (sym->st_shndx == SHN_UNDEF) &&
                    (ELF_ST_BIND(sym->st_info) != STB_WEAK)) {
                        if (vis && (vis != STV_SINGLETON)) {
                                sym_undef_entry(ofl, sdp, BNDLOCAL,
                                    FLG_OF_FATAL, &undef_state);
                                continue;
                        }
                }

                /*
                 * If this symbol is defined in a non-allocatable section,
                 * reduce it to local symbol.
                 */
                if (((isp = sdp->sd_isc) != 0) && isp->is_shdr &&
                    ((isp->is_shdr->sh_flags & SHF_ALLOC) == 0)) {
                        sdp->sd_flags |= (FLG_SY_REDUCED | FLG_SY_HIDDEN);
                }

                /*
                 * If this symbol originated as a SHN_SUNW_IGNORE, it will have
                 * been processed as an SHN_UNDEF.  Return the symbol to its
                 * original index for validation, and propagation to the output
                 * file.
                 */
                if (sdp->sd_flags & FLG_SY_IGNORE)
                        sdp->sd_shndx = SHN_SUNW_IGNORE;

                if (undef) {
                        /*
                         * If a non-weak reference remains undefined, or if a
                         * mapfile reference is not bound to the relocatable
                         * objects that make up the object being built, we have
                         * a fatal error.
                         *
                         * The exceptions are symbols which are defined to be
                         * found in the parent (FLG_SY_PARENT), which is really
                         * only meaningful for direct binding, or are defined
                         * external (FLG_SY_EXTERN) so as to suppress -zdefs
                         * errors.
                         *
                         * Register symbols are always allowed to be UNDEF.
                         *
                         * Note that we don't include references created via -u
                         * in the same shared object binding test.  This is for
                         * backward compatibility, in that a number of archive
                         * makefile rules used -u to cause archive extraction.
                         * These same rules have been cut and pasted to apply
                         * to shared objects, and thus although the -u reference
                         * is redundant, flagging it as fatal could cause some
                         * build to fail.  Also we have documented the use of
                         * -u as a mechanism to cause binding to weak version
                         * definitions, thus giving users an error condition
                         * would be incorrect.
                         */
                        if (!(sdp->sd_flags & FLG_SY_REGSYM) &&
                            ((sym->st_shndx == SHN_UNDEF) &&
                            ((ELF_ST_BIND(sym->st_info) != STB_WEAK) &&
                            ((sdp->sd_flags &
                            (FLG_SY_PARENT | FLG_SY_EXTERN)) == 0)) ||
                            ((sdp->sd_flags &
                            (FLG_SY_MAPREF | FLG_SY_MAPUSED | FLG_SY_HIDDEN |
                            FLG_SY_PROTECT)) == FLG_SY_MAPREF))) {
                                sym_undef_entry(ofl, sdp, UNDEF, undef,
                                    &undef_state);
                                undeferr = 1;
                        }

                } else {
                        /*
                         * For building things like shared objects (or anything
                         * -znodefs), undefined symbols are allowed.
                         *
                         * If a mapfile reference remains undefined the user
                         * would probably like a warning at least (they've
                         * usually mis-spelt the reference).  Refer to the above
                         * comments for discussion on -u references, which
                         * are not tested for in the same manner.
                         */
                        if ((sdp->sd_flags &
                            (FLG_SY_MAPREF | FLG_SY_MAPUSED)) ==
                            FLG_SY_MAPREF) {
                                sym_undef_entry(ofl, sdp, UNDEF, FLG_OF_WARN,
                                    &undef_state);
                                undeferr = 1;
                        }
                }

                /*
                 * If this symbol comes from a dependency mark the dependency
                 * as required (-z ignore can result in unused dependencies
                 * being dropped).  If we need to record dependency versioning
                 * information indicate what version of the needed shared object
                 * this symbol is part of.  Flag the symbol as undefined if it
                 * has not been made available to us.
                 */
                if ((sdp->sd_ref == REF_DYN_NEED) &&
                    (!(sdp->sd_flags & FLG_SY_REFRSD))) {
                        sdp->sd_file->ifl_flags |= FLG_IF_DEPREQD;

                        /*
                         * Capture that we've bound to a symbol that doesn't
                         * allow being directly bound to.
                         */
                        if (sdp->sd_flags & FLG_SY_NDIR)
                                ofl->ofl_flags1 |= FLG_OF1_NGLBDIR;

                        if (sdp->sd_file->ifl_vercnt) {
                                int             vndx;
                                Ver_index       *vip;

                                vndx = sdp->sd_aux->sa_dverndx;
                                vip = &sdp->sd_file->ifl_verndx[vndx];
                                if (vip->vi_flags & FLG_VER_AVAIL) {
                                        vip->vi_flags |= FLG_VER_REFER;
                                } else {
```

```
                                        sym_undef_entry(ofl, sdp, NOTAVAIL,
                                            FLG_OF_FATAL, &undef_state);
                                        continue;
                                }
                        }
                }

                /*
                 * Test that we do not bind to symbol supplied from an implicit
                 * shared object.  If a binding is from a weak reference it can
                 * be ignored.
                 */
                if (needed && !undeferr && (sdp->sd_flags & FLG_SY_GLOBREF) &&
                    (sdp->sd_ref == REF_DYN_NEED) &&
                    (sdp->sd_flags & FLG_SY_NOTAVAIL)) {
                        sym_undef_entry(ofl, sdp, IMPLICIT, needed,
                            &undef_state);
                        if (needed == FLG_OF_FATAL)
                                continue;
                }

                /*
                 * Test that a symbol isn't going to be reduced to local scope
                 * which actually wants to bind to a shared object - if so it's
                 * a fatal error.
                 */
                if ((sdp->sd_ref == REF_DYN_NEED) &&
                    (sdp->sd_flags & (FLG_SY_HIDDEN | FLG_SY_PROTECT))) {
                        sym_undef_entry(ofl, sdp, BNDLOCAL, FLG_OF_FATAL,
                            &undef_state);
                        continue;
                }

                /*
                 * If the output image is to be versioned then all symbol
                 * definitions must be associated with a version.  Remove any
                 * versioning that might be left associated with an undefined
                 * symbol.
                 */
                if (verdesc && (sdp->sd_ref == REF_REL_NEED)) {
                        if (sym->st_shndx == SHN_UNDEF) {
                                if (sdp->sd_aux && sdp->sd_aux->sa_overndx)
                                        sdp->sd_aux->sa_overndx = 0;
                        } else {
                                if (!SYM_IS_HIDDEN(sdp) && sdp->sd_aux &&
                                    (sdp->sd_aux->sa_overndx == 0)) {
                                        sym_undef_entry(ofl, sdp, NOVERSION,
                                            verdesc, &undef_state);
                                        continue;
                                }
                        }
                }

                /*
                 * If we don't need the symbol there's no need to process it
                 * any further.
                 */
                if (sdp->sd_ref == REF_DYN_SEEN)
                        continue;

                /*
                 * Calculate the size and alignment requirements for the global
                 * .bss and .tls sections.  If we're building a relocatable
                 * object only account for scoped COMMON symbols (these will
                 * be converted to .bss references).
                 *
```

```
		 * When -z nopartial is in effect, partially initialized
		 * symbols are directed to the special .data section
		 * created for that purpose (ofl->ofl_isparexpn).
		 * Otherwise, partially initialized symbols go to .bss.
		 *
		 * Also refer to make_mvsections() in sunwmove.c
		 */
		if ((sym->st_shndx == SHN_COMMON) &&
		    (((oflags & FLG_OF_RELOBJ) == 0) ||
		    (SYM_IS_HIDDEN(sdp) && (oflags & FLG_OF_PROCRED)))) {
			if ((sdp->sd_move == NULL) ||
			    ((sdp->sd_flags & FLG_SY_PAREXPN) == 0)) {
				if (type != STT_TLS) {
					need_bss = TRUE;
					bsssize = (Xword)S_ROUND(bsssize,
					    sym->st_value) + sym->st_size;
					if (sym->st_value > bssalign)
						bssalign = sym->st_value;
				} else {
					need_tlsbss = TRUE;
					tlssize = (Xword)S_ROUND(tlssize,
					    sym->st_value) + sym->st_size;
					if (sym->st_value > tlsalign)
						tlsalign = sym->st_value;
				}
			}
		}

#if	defined(_ELF64)
		/*
		 * Calculate the size and alignment requirement for the global
		 * .lbss. TLS or partially initialized symbols do not need to be
		 * considered yet.
		 */
		if ((ld_targ.t_m.m_mach == EM_AMD64) &&
		    (sym->st_shndx == SHN_X86_64_LCOMMON)) {
			need_lbss = TRUE;
			lbsssize = (Xword)S_ROUND(lbsssize, sym->st_value) +
			    sym->st_size;
			if (sym->st_value > lbssalign)
				lbssalign = sym->st_value;
		}
#endif
		/*
		 * If a symbol was referenced via the command line
		 * (ld -u <>, ...), then this counts as a reference against the
		 * symbol. Mark any section that symbol is defined in.
		 */
		if (((isp = sdp->sd_isc) != 0) &&
		    (sdp->sd_flags & FLG_SY_CMDREF)) {
			isp->is_flags |= FLG_IS_SECTREF;
			isp->is_file->ifl_flags |= FLG_IF_FILEREF;
		}

		/*
		 * Update the symbol count and the associated name string size.
		 * Note, a capabilities symbol must remain as visible as a
		 * global symbol.  However, the runtime linker recognizes the
		 * hidden requirement and ensures the symbol isn't made globally
		 * available at runtime.
		 */
		if (SYM_IS_HIDDEN(sdp) && (oflags & FLG_OF_PROCRED)) {
			/*
			 * If any reductions are being processed, keep a count
			 * of eliminated symbols, and if the symbol is being
			 * reduced to local, count it's size for the .symtab.
```

```
			 */
			if (sdp->sd_flags & FLG_SY_ELIM) {
				ofl->ofl_elimcnt++;
			} else {
				ofl->ofl_scopecnt++;
				if ((((sdp->sd_flags & FLG_SY_REGSYM) == 0) ||
				    sym->st_name) && (st_insert(ofl->ofl_strtab,
				    sdp->sd_name) == -1))
					return (S_ERROR);
				if (allow_ldynsym && sym->st_name &&
				    ldynsym_symtype[type]) {
					ofl->ofl_dynscopecnt++;
					if (st_insert(ofl->ofl_dynstrtab,
					    sdp->sd_name) == -1)
						return (S_ERROR);
					/* Include it in sort section? */
					DYNSORT_COUNT(sdp, sym, type, ++);
				}
			}
		} else {
			ofl->ofl_globcnt++;

			/*
			 * Check to see if this global variable should go into
			 * a sort section. Sort sections require a
			 * .SUNW_ldynsym section, so, don't check unless a
			 * .SUNW_ldynsym is allowed.
			 */
			if (allow_ldynsym)
				DYNSORT_COUNT(sdp, sym, type, ++);

			/*
			 * If global direct bindings are in effect, or this
			 * symbol has bound to a dependency which was specified
			 * as requiring direct bindings, and it hasn't
			 * explicitly been defined as a non-direct binding
			 * symbol, mark it.
			 */
			if (((ofl->ofl_dtflags_1 & DF_1_DIRECT) || (isp &&
			    (isp->is_file->ifl_flags & FLG_IF_DIRECT))) &&
			    ((sdp->sd_flags & FLG_SY_NDIR) == 0))
				sdp->sd_flags |= FLG_SY_DIR;

			/*
			 * Insert the symbol name.
			 */
			if (((sdp->sd_flags & FLG_SY_REGSYM) == 0) ||
			    sym->st_name) {
				if (st_insert(ofl->ofl_strtab,
				    sdp->sd_name) == -1)
					return (S_ERROR);

				if (!(ofl->ofl_flags & FLG_OF_RELOBJ) &&
				    (st_insert(ofl->ofl_dynstrtab,
				    sdp->sd_name) == -1))
					return (S_ERROR);
			}

			/*
			 * If this section offers a global symbol - record that
			 * fact.
			 */
			if (isp) {
				isp->is_flags |= FLG_IS_SECTREF;
				isp->is_file->ifl_flags |= FLG_IF_FILEREF;
			}
		}
	}

	/*
	 * Guidance: Use -z defs|nodefs when building shared objects.
	 *
	 * Our caller issues this, unless we mask it out here. So we mask it
	 * out unless we've issued at least one warnings or fatal error.
	 */
	if (!((oflags & FLG_OF_SHAROBJ) && OFL_GUIDANCE(ofl, FLG_OFG_NO_DEFS) &&
	    (undef_state & (FLG_OF_FATAL | FLG_OF_WARN))))
		ofl->ofl_guideflags |= FLG_OFG_NO_DEFS;

	/*
	 * If we've encountered a fatal error during symbol validation then
	 * return now.
	 */
	if (ofl->ofl_flags & FLG_OF_FATAL)
		return (1);

	/*
	 * Now that symbol resolution is completed, scan any register symbols.
	 * From now on, we're only interested in those that contribute to the
	 * output file.
	 */
	if (ofl->ofl_regsyms) {
		int	ndx;

		for (ndx = 0; ndx < ofl->ofl_regsymsno; ndx++) {
			if ((sdp = ofl->ofl_regsyms[ndx]) == NULL)
				continue;
			if (sdp->sd_ref != REF_REL_NEED) {
				ofl->ofl_regsyms[ndx] = NULL;
				continue;
			}

			ofl->ofl_regsymcnt++;
			if (sdp->sd_sym->st_name == 0)
				sdp->sd_name = MSG_ORIG(MSG_STR_EMPTY);

			if (SYM_IS_HIDDEN(sdp) ||
			    (ELF_ST_BIND(sdp->sd_sym->st_info) == STB_LOCAL))
				ofl->ofl_lregsymcnt++;
		}
	}

	/*
	 * Generate the .bss section now that we know its size and alignment.
	 */
	if (need_bss) {
		if (ld_make_bss(ofl, bsssize, bssalign,
		    ld_targ.t_id.id_bss) == S_ERROR)
			return (S_ERROR);
	}
	if (need_tlsbss) {
		if (ld_make_bss(ofl, tlssize, tlsalign,
		    ld_targ.t_id.id_tlsbss) == S_ERROR)
			return (S_ERROR);
	}
#if	defined(_ELF64)
	if ((ld_targ.t_m.m_mach == EM_AMD64) &&
	    need_lbss && !(oflags & FLG_OF_RELOBJ)) {
		if (ld_make_bss(ofl, lbsssize, lbssalign,
		    ld_targ.t_id.id_lbss) == S_ERROR)
			return (S_ERROR);
	}
```

```
#endif
	/*
	 * Determine what entry point symbol we need, and if found save its
	 * symbol descriptor so that we can update the ELF header entry with the
	 * symbols value later (see update_oehdr).  Make sure the symbol is
	 * tagged to ensure its update in case -s is in effect.  Use any -e
	 * option first, or the default entry points '_start' and 'main'.
	 */
	ret = 0;
	if (ofl->ofl_entry) {
		if ((sdp = ld_sym_find(ofl->ofl_entry, SYM_NOHASH,
		    NULL, ofl)) == NULL) {
			ld_eprintf(ofl, ERR_FATAL, MSG_INTL(MSG_ARG_NOENTRY),
			    ofl->ofl_entry);
			ret++;
		} else if (ensure_sym_local(ofl, sdp,
		    MSG_INTL(MSG_SYM_ENTRY)) != 0) {
			ret++;
		} else {
			ofl->ofl_entry = (void *)sdp;
		}
	} else if (((sdp = ld_sym_find(MSG_ORIG(MSG_SYM_START),
	    SYM_NOHASH, NULL, ofl)) != NULL) && (ensure_sym_local(ofl,
	    sdp, 0) == 0)) {
		ofl->ofl_entry = (void *)sdp;

	} else if (((sdp = ld_sym_find(MSG_ORIG(MSG_SYM_MAIN),
	    SYM_NOHASH, NULL, ofl)) != NULL) && (ensure_sym_local(ofl,
	    sdp, 0) == 0)) {
		ofl->ofl_entry = (void *)sdp;
	}

	/*
	 * If ld -zdtrace=<sym> was given, then validate that the symbol is
	 * defined within the current object being built.
	 */
	if ((sdp = ofl->ofl_dtracesym) != 0)
		ret += ensure_sym_local(ofl, sdp, MSG_ORIG(MSG_STR_DTRACE));

	/*
	 * If any initarray, finiarray or preinitarray functions have been
	 * requested, make sure they are defined within the current object
	 * being built.
	 */
	if (ofl->ofl_initarray) {
		ret += ensure_array_local(ofl, ofl->ofl_initarray,
		    MSG_ORIG(MSG_SYM_INITARRAY));
	}
	if (ofl->ofl_finiarray) {
		ret += ensure_array_local(ofl, ofl->ofl_finiarray,
		    MSG_ORIG(MSG_SYM_FINIARRAY));
	}
	if (ofl->ofl_preiarray) {
		ret += ensure_array_local(ofl, ofl->ofl_preiarray,
		    MSG_ORIG(MSG_SYM_PREINITARRAY));
	}

	if (ret)
		return (S_ERROR);

	/*
	 * If we're required to record any needed dependencies versioning
	 * information calculate it now that all symbols have been validated.
	 */
	if ((oflags & (FLG_OF_VERNEED | FLG_OF_NOVERSEC)) == FLG_OF_VERNEED)
		return (ld_vers_check_need(ofl));
	else
		return (1);
}

/*
 * qsort(3c) comparison function.  As an optimization for associating weak
 * symbols to their strong counterparts sort global symbols according to their
 * section index, address and binding.
 */
static int
compare(const void *sdpp1, const void *sdpp2)
{
	Sym_desc	*sdp1 = *((Sym_desc **)sdpp1);
	Sym_desc	*sdp2 = *((Sym_desc **)sdpp2);
	Sym		*sym1, *sym2;
	uchar_t		bind1, bind2;

	/*
	 * Symbol descriptors may be zero, move these to the front of the
	 * sorted array.
	 */
	if (sdp1 == NULL)
		return (-1);
	if (sdp2 == NULL)
		return (1);

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

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

	/*
	 * Compare the symbols value (address).
	 */
	if (sym1->st_value > sym2->st_value)
		return (1);
	if (sym1->st_value < sym2->st_value)
		return (-1);

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

	/*
	 * If two symbols have the same address place the weak symbol before
	 * any strong counterpart.
	 */
	if (bind1 > bind2)
		return (-1);
	if (bind1 < bind2)
		return (1);

	return (0);
}

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

```
 * is issued when a symbol address/size is not contained by the
 * target section.
 *
 * Such objects are at least partially corrupt, and the user would
 * be well advised to be skeptical of them, and to ask their compiler
 * supplier to fix the problem. However, a distinction needs to be
 * made between symbols that reference readonly text, and those that
 * access writable data. Other than throwing off profiling results,
 * the readonly section case is less serious. We have encountered
 * such objects in the field. In order to allow existing objects
 * to continue working, we issue a warning rather than a fatal error
 * if the symbol is against readonly text. Other cases are fatal.
 */
static void
issue_badaddr_msg(Ifl_desc *ifl, Ofl_desc *ofl, Sym_desc *sdp,
    Sym *sym, Word shndx)
{
	Error		err;
	const char	*msg;

	if ((sdp->sd_isc->is_shdr->sh_flags & (SHF_WRITE | SHF_ALLOC)) ==
	    SHF_ALLOC) {
		msg = MSG_INTL(MSG_SYM_BADADDR_ROTXT);
		err = ERR_WARNING;
	} else {
		msg = MSG_INTL(MSG_SYM_BADADDR);
		err = ERR_FATAL;
	}

	ld_eprintf(ofl, err, msg, demangle(sdp->sd_name),
	    ifl->ifl_name, shndx, sdp->sd_isc->is_name,
	    EC_XWORD(sdp->sd_isc->is_shdr->sh_size),
	    EC_XWORD(sym->st_value), EC_XWORD(sym->st_size));
}

/*
 * Global symbols that are candidates for translation to local capability
 * symbols under -z symbolcap, are maintained on a local symbol list.  Once
 * all symbols of a file are processed, this list is traversed to cull any
 * unnecessary weak symbol aliases.
 */
typedef struct {
	Sym_desc	*c_nsdp;	/* new lead symbol */
	Sym_desc	*c_osdp;	/* original symbol */
	Cap_group	*c_group;	/* symbol capability group */
	Word		c_ndx;		/* symbol index */
} Cap_pair;

/*
 * Process the symbol table for the specified input file.  At this point all
 * input sections from this input file have been assigned an input section
 * descriptor which is saved in the ‘ifl_isdesc’ array.
 *
 *  -	local symbols are saved (as is) if the input file is a  relocatable
 *	object
 *
 *  -	global symbols are added to the linkers internal symbol table if they
 *	are not already present, otherwise a symbol resolution function is
 *	called upon to resolve the conflict.
 */
uintptr_t
ld_sym_process(Is_desc *isc, Ifl_desc *ifl, Ofl_desc *ofl)
{
	/*
	 * This macro tests the given symbol to see if it is out of
	 * range relative to the section it references.
```

```
         *
         * entry:
         *      - ifl is a relative object (ET_REL)
         *      _sdp - Symbol descriptor
         *      _sym - Symbol
         *      _type - Symbol type
         *
         * The following are tested:
         *      - Symbol length is non-zero
         *      - Symbol type is a type that references code or data
         *      - Referenced section is not 0 (indicates an UNDEF symbol)
         *        and is not in the range of special values above SHN_LORESERVE
         *        (excluding SHN_XINDEX, which is OK).
         *      - We have a valid section header for the target section
         *
         * If the above are all true, and the symbol position is not
         * contained by the target section, this macro evaluates to
         * True (1). Otherwise, False(0).
         */
#define SYM_LOC_BADADDR(_sdp, _sym, _type) \
        (_sym->st_size && dynsymsort_symtype[_type] && \
        (_sym->st_shndx != SHN_UNDEF) && \
        ((_sym->st_shndx < SHN_LORESERVE) || \
                (_sym->st_shndx == SHN_XINDEX)) && \
        _sdp->sd_isc && _sdp->sd_isc->is_shdr && \
        ((_sym->st_value + _sym->st_size) > _sdp->sd_isc->is_shdr->sh_size))

        Conv_inv_buf_t  inv_buf;
        Sym             *sym = (Sym *)isc->is_indata->d_buf;
        Word            *symshndx = NULL;
        Shdr            *shdr = isc->is_shdr;
        Sym_desc        *sdp;
        size_t          strsize;
        char            *strs;
        uchar_t         type, bind;
        Word            ndx, hash, local, total;
        uchar_t         osabi = ifl->ifl_ehdr->e_ident[EI_OSABI];
        Half            mach = ifl->ifl_ehdr->e_machine;
        Half            etype = ifl->ifl_ehdr->e_type;
        int             etype_rel;
        const char      *symsecname, *strsecname;
        Word            symsecndx;
        avl_index_t     where;
        int             test_gnu_hidden_bit, weak;
        Cap_desc        *cdp = NULL;
        Alist           *cappairs = NULL;

        /*
         * Its possible that a file may contain more that one symbol table,
         * ie. .dynsym and .symtab in a shared library.  Only process the first
         * table (here, we assume .dynsym comes before .symtab).
         */
        if (ifl->ifl_symscnt)
                return (1);

        if (isc->is_symshndx)
                symshndx = isc->is_symshndx->is_indata->d_buf;

        DBG_CALL(Dbg_syms_process(ofl->ofl_lml, ifl));

        symsecndx = isc->is_scnndx;
        if (isc->is_name)
                symsecname = isc->is_name;
        else
                symsecname = MSG_ORIG(MSG_STR_EMPTY);
```

```
        /*
         * From the symbol tables section header information determine which
         * strtab table is needed to locate the actual symbol names.
         */
        if (ifl->ifl_flags & FLG_IF_HSTRTAB) {
                ndx = shdr->sh_link;
                if ((ndx == 0) || (ndx >= ifl->ifl_shnum)) {
                        ld_eprintf(ofl, ERR_FATAL,
                            MSG_INTL(MSG_FIL_INVSHLINK), ifl->ifl_name,
                            EC_WORD(symsecndx), symsecname, EC_XWORD(ndx));
                        return (S_ERROR);
                }
                strsize = ifl->ifl_isdesc[ndx]->is_shdr->sh_size;
                strs = ifl->ifl_isdesc[ndx]->is_indata->d_buf;
                if (ifl->ifl_isdesc[ndx]->is_name)
                        strsecname = ifl->ifl_isdesc[ndx]->is_name;
                else
                        strsecname = MSG_ORIG(MSG_STR_EMPTY);
        } else {
                /*
                 * There is no string table section in this input file
                 * although there are symbols in this symbol table section.
                 * This means that these symbols do not have names.
                 * Currently, only scratch register symbols are allowed
                 * not to have names.
                 */
                strsize = 0;
                strs = (char *)MSG_ORIG(MSG_STR_EMPTY);
                strsecname = MSG_ORIG(MSG_STR_EMPTY);
        }

        /*
         * Determine the number of local symbols together with the total
         * number we have to process.
         */
        total = (Word)(shdr->sh_size / shdr->sh_entsize);
        local = shdr->sh_info;

        /*
         * Allocate a symbol table index array and a local symbol array
         * (global symbols are processed and added to the ofl->ofl_symbkt[]
         * array).  If we are dealing with a relocatable object, allocate the
         * local symbol descriptors.  If this isn't a relocatable object we
         * still have to process any shared object locals to determine if any
         * register symbols exist.  Although these aren't added to the output
         * image, they are used as part of symbol resolution.
         */
        if ((ifl->ifl_oldndx = libld_malloc((size_t)(total *
            sizeof (Sym_desc *)))) == NULL)
                return (S_ERROR);
        etype_rel = (etype == ET_REL);
        if (etype_rel && local) {
                if ((ifl->ifl_locs =
                    libld_calloc(sizeof (Sym_desc), local)) == NULL)
                        return (S_ERROR);
                /* LINTED */
                ifl->ifl_locscnt = (Word)local;
        }
        ifl->ifl_symscnt = total;

        /*
         * If there are local symbols to save add them to the symbol table
         * index array.
         */
        if (local) {
                int             allow_ldynsym = OFL_ALLOW_LDYNSYM(ofl);
                Sym_desc        *last_file_sdp = NULL;
                int             last_file_ndx = 0;

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

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

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

                        /*
                         * Now that we have the name, if the section index
                         * was bad, report it.
                         */
                        if (shndx_bad) {
                                ld_eprintf(ofl, ERR_WARNING,
                                    MSG_INTL(MSG_SYM_INVSHNDX),
                                    demangle_symname(name, symsecname, ndx),
                                    ifl->ifl_name,
                                    conv_sym_shndx(osabi, mach, sym->st_shndx,
                                    CONV_FMT_DECIMAL, &inv_buf));
                                continue;
                        }

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

```
                                         * the error message and move on.
                                         */
                                        (void) (*ld_targ.t_ms.ms_reg_check)
                                            (rsdp, sym, name, ifl, ofl);
                                        continue;
                                }

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

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

                        /*
                         * Fill in the remaining symbol descriptor information.
                         */
                        if (sdp == NULL) {
                                sdp = &(ifl->ifl_locs[ndx]);
                                sdp->sd_ref = REF_REL_NEED;
                                sdp->sd_symndx = ndx;
                        }
                        if (rsdp == NULL) {
                                sdp->sd_name = name;
                                sdp->sd_sym = sym;
                                sdp->sd_shndx = shndx;
                                sdp->sd_flags = sdflags;
                                sdp->sd_file = ifl;
                                ifl->ifl_oldndx[ndx] = sdp;
                        }

                        DBG_CALL(Dbg_syms_entry(ofl->ofl_lml, ndx, sdp));

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

                        /*
                         * Process any register symbols.
                         */
                        if (sdp->sd_flags & FLG_SY_REGSYM) {
                                /*
                                 * Add a diagnostic to indicate we've caught a
                                 * register symbol, as this can be useful if a
                                 * register conflict is later discovered.
                                 */
                                DBG_CALL(Dbg_syms_entered(ofl, sym, sdp));

                                /*
                                 * If this register symbol hasn't already been
                                 * recorded, enter it now.
                                 *
                                 * The presence of FLG_SY_REGSYM means that
                                 * the pointers in ld_targ.t_ms are non-NULL.
                                 */
                                if ((rsdp == NULL) &&
					((*ld_targ.t_ms.ms_reg_enter)(sdp, ofl) ==
					0))
						return (S_ERROR);
			}

			/*
			 * Assign an input section.
			 */
			if ((sym->st_shndx != SHN_UNDEF) &&
			    ((sdp->sd_flags & FLG_SY_SPECSEC) == 0))
				sdp->sd_isc = ifl->ifl_isdesc[shndx];

			/*
			 * If this symbol falls within the range of a section
			 * being discarded, then discard the symbol itself.
			 * There is no reason to keep this local symbol.
			 */
			if (sdp->sd_isc &&
			    (sdp->sd_isc->is_flags & FLG_IS_DISCARD)) {
				sdp->sd_flags |= FLG_SY_ISDISC;
				DBG_CALL(Dbg_syms_discarded(ofl->ofl_lml, sdp));
				continue;
			}

			/*
			 * Skip any section symbols as new versions of these
			 * will be created.
			 */
			if ((type = ELF_ST_TYPE(sym->st_info)) == STT_SECTION) {
				if (sym->st_shndx == SHN_UNDEF) {
					ld_eprintf(ofl, ERR_WARNING,
					    MSG_INTL(MSG_SYM_INVSHNDX),
					    demangle_symname(name, symsecname,
					    ndx), ifl->ifl_name,
					    conv_sym_shndx(osabi, mach,
					    sym->st_shndx, CONV_FMT_DECIMAL,
					    &inv_buf));
				}
				continue;
			}

			/*
			 * For a relocatable object, if this symbol is defined
			 * and has non-zero length and references an address
			 * within an associated section, then check its extents
			 * to make sure the section boundaries encompass it.
			 * If they don't, the ELF file is corrupt.
			 */
			if (etype_rel) {
				if (SYM_LOC_BADADDR(sdp, sym, type)) {
					issue_badaddr_msg(ifl, ofl, sdp,
					    sym, shndx);
					if (ofl->ofl_flags & FLG_OF_FATAL)
						continue;
				}

				/*
				 * We have observed relocatable objects
				 * containing identical adjacent STT_FILE
				 * symbols. Discard any other than the first,
				 * as they are all equivalent and the extras
				 * do not add information.
				 *
				 * For the purpose of this test, we assume
				 * that only the symbol type and the string
				 * table offset (st_name) matter.
```

```
				 */
				if (type == STT_FILE) {
					int toss = (last_file_sdp != NULL) &&
					    ((ndx - 1) == last_file_ndx) &&
					    (sym->st_name ==
					    last_file_sdp->sd_sym->st_name);

					last_file_sdp = sdp;
					last_file_ndx = ndx;
					if (toss) {
						sdp->sd_flags |= FLG_SY_INVALID;
						DBG_CALL(Dbg_syms_dup_discarded(
						    ofl->ofl_lml, ndx, sdp));
						continue;
					}
				}
			}


			/*
			 * Sanity check for TLS
			 */
			if ((sym->st_size != 0) && ((type == STT_TLS) &&
			    (sym->st_shndx != SHN_COMMON))) {
				Is_desc *isp = sdp->sd_isc;

				if ((isp == NULL) || (isp->is_shdr == NULL) ||
				    ((isp->is_shdr->sh_flags & SHF_TLS) == 0)) {
					ld_eprintf(ofl, ERR_FATAL,
					    MSG_INTL(MSG_SYM_TLS),
					    demangle(sdp->sd_name),
					    ifl->ifl_name);
					continue;
				}
			}

			/*
			 * Carry our some basic sanity checks (these are just
			 * some of the erroneous symbol entries we've come
			 * across, there's probably a lot more).  The symbol
			 * will not be carried forward to the output file, which
			 * won't be a problem unless a relocation is required
			 * against it.
			 */
			if (((sdp->sd_flags & FLG_SY_SPECSEC) &&
			    ((sym->st_shndx == SHN_COMMON)) ||
			    ((type == STT_FILE) &&
			    (sym->st_shndx != SHN_ABS))) ||
			    (sdp->sd_isc && (sdp->sd_isc->is_osdesc == NULL))) {
				ld_eprintf(ofl, ERR_WARNING,
				    MSG_INTL(MSG_SYM_INVSHNDX),
				    demangle_symname(name, symsecname, ndx),
				    ifl->ifl_name,
				    conv_sym_shndx(osabi, mach, sym->st_shndx,
				    CONV_FMT_DECIMAL, &inv_buf));
				sdp->sd_isc = NULL;
				sdp->sd_flags |= FLG_SY_INVALID;
				continue;
			}

			/*
			 * As these local symbols will become part of the output
			 * image, record their number and name string size.
			 * Globals are counted after all input file processing
			 * (and hence symbol resolution) is complete during
			 * sym_validate().
```

```
                         */
                        if (!(ofl->ofl_flags & FLG_OF_REDLSYM) &&
                            symtab_enter) {
                                ofl->ofl_locscnt++;

                                if ((((sdp->sd_flags & FLG_SY_REGSYM) == 0) ||
                                    sym->st_name) && (st_insert(ofl->ofl_strtab,
                                    sdp->sd_name) == -1))
                                        return (S_ERROR);

                                if (allow_ldynsym && sym->st_name &&
                                    ldynsym_symtype[type]) {
                                        ofl->ofl_dynlocscnt++;
                                        if (st_insert(ofl->ofl_dynstrtab,
                                            sdp->sd_name) == -1)
                                                return (S_ERROR);
                                        /* Include it in sort section? */
                                        DYNSORT_COUNT(sdp, sym, type, ++);
                                }
                        }
                }
        }

        /*
         * The GNU ld interprets the top bit of the 16-bit Versym value
         * (0x8000) as the "hidden" bit. If this bit is set, the linker
         * is supposed to act as if that symbol does not exist. The Solaris
         * linker does not support this mechanism, or the model of interface
         * evolution that it allows, but we honor it in GNU ld produced
         * objects in order to interoperate with them.
         *
         * Determine if we should honor the GNU hidden bit for this file.
         */
        test_gnu_hidden_bit = ((ifl->ifl_flags & FLG_IF_GNUVER) != 0) &&
            (ifl->ifl_versym != NULL);

        /*
         * Determine whether object capabilities for this file are being
         * converted into symbol capabilities.  If so, global function symbols,
         * and initialized global data symbols, need special translation and
         * processing.
         */
        if ((etype == ET_REL) && (ifl->ifl_flags & FLG_IF_OTOSCAP))
                cdp = ifl->ifl_caps;

        /*
         * Now scan the global symbols entering them in the internal symbol
         * table or resolving them as necessary.
         */
        sym = (Sym *)isc->is_indata->d_buf;
        sym += local;
        weak = 0;
        /* LINTED */
        for (ndx = (int)local; ndx < total; sym++, ndx++) {
                const char      *name;
                sd_flag_t       sdflags = 0;
                Word            shndx;
                int             shndx_bad = 0;
                Sym             *nsym = sym;
                Cap_pair        *cpp = NULL;
                uchar_t         ntype;

                /*
                 * Determine and validate the associated section index.
                 */
                if (symshndx && (nsym->st_shndx == SHN_XINDEX)) {
```

```
                        shndx = symshndx[ndx];
                } else if ((shndx = nsym->st_shndx) >= SHN_LORESERVE) {
                        sdflags |= FLG_SY_SPECSEC;
                } else if (shndx > ifl->ifl_shnum) {
                        /* We need the name before we can issue error */
                        shndx_bad = 1;
                }

                /*
                 * Check if st_name has a valid value or not.
                 */
                if ((name = string(ofl, ifl, nsym, strs, strsize, ndx, shndx,
                    symsecndx, symsecname, strsecname, &sdflags)) == NULL)
                        continue;

                /*
                 * Now that we have the name, report an erroneous section index.
                 */
                if (shndx_bad) {
                        ld_eprintf(ofl, ERR_WARNING, MSG_INTL(MSG_SYM_INVSHNDX),
                            demangle_symname(name, symsecname, ndx),
                            ifl->ifl_name,
                            conv_sym_shndx(osabi, mach, nsym->st_shndx,
                            CONV_FMT_DECIMAL, &inv_buf));
                        continue;
                }

                /*
                 * Test for the GNU hidden bit, and ignore symbols that
                 * have it set.
                 */
                if (test_gnu_hidden_bit &&
                    ((ifl->ifl_versym[ndx] & 0x8000) != 0))
                        continue;

                /*
                 * The linker itself will generate symbols for _end, _etext,
                 * _edata, _DYNAMIC and _PROCEDURE_LINKAGE_TABLE_, so don't
                 * bother entering these symbols from shared objects.  This
                 * results in some wasted resolution processing, which is hard
                 * to feel, but if nothing else, pollutes diagnostic relocation
                 * output.
                 */
                if (name[0] && (etype == ET_DYN) && (nsym->st_size == 0) &&
                    (ELF_ST_TYPE(nsym->st_info) == STT_OBJECT) &&
                    (name[0] == '_') && ((name[1] == 'e') ||
                    (name[1] == 'D') || (name[1] == 'P')) &&
                    ((strcmp(name, MSG_ORIG(MSG_SYM_ETEXT_U)) == 0) ||
                    (strcmp(name, MSG_ORIG(MSG_SYM_EDATA_U)) == 0) ||
                    (strcmp(name, MSG_ORIG(MSG_SYM_END_U)) == 0) ||
                    (strcmp(name, MSG_ORIG(MSG_SYM_DYNAMIC_U)) == 0) ||
                    (strcmp(name, MSG_ORIG(MSG_SYM_PLKTBL_U)) == 0))) {
                        ifl->ifl_oldndx[ndx] = 0;
                        continue;
                }

                /*
                 * The '-z wrap=XXX' option emulates the GNU ld --wrap=XXX
                 * option. When XXX is the symbol to be wrapped:
                 *
                 *  -   An undefined reference to XXX is converted to __wrap_XXX
                 *  -   An undefined reference to __real_XXX is converted to XXX
                 *
                 * The idea is that the user can supply a wrapper function
                 * __wrap_XXX that does some work, and then uses the name
                 * __real_XXX to pass the call on to the real function. The
```

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

			/*
			 * If this is the __real_XXX form, advance the
			 * pointer to reference the wrapped name.
			 */
			wsn.wsn_name = name;
			if ((*name == '_') &&
			    (strncmp(name, MSG_ORIG(MSG_STR_UU_REAL_U),
			    MSG_STR_UU_REAL_U_SIZE) == 0))
				wsn.wsn_name += MSG_STR_UU_REAL_U_SIZE;

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

				name = (wsn.wsn_name == name) ?
				    wsnp->wsn_wrapname : wsn.wsn_name;
				DBG_CALL(Dbg_syms_wrap(ofl->ofl_lml, ndx,
				    old_name, name));
			}
		}

		/*
		 * Determine and validate the symbols binding.
		 */
		bind = ELF_ST_BIND(nsym->st_info);
		if ((bind != STB_GLOBAL) && (bind != STB_WEAK)) {
			ld_eprintf(ofl, ERR_WARNING, MSG_INTL(MSG_SYM_NONGLOB),
			    demangle_symname(name, symsecname, ndx),
			    ifl->ifl_name,
			    conv_sym_info_bind(bind, 0, &inv_buf));
			continue;
		}
		if (bind == STB_WEAK)
			weak++;

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

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

```
				    CONV_FMT_DECIMAL, &inv_buf));
				continue;
			}

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

				/*
				 * Create a dummy symbol entry so that if we
				 * find any references to this discarded symbol
				 * we can compensate.
				 */
				sdp->sd_name = name;
				sdp->sd_sym = nsym;
				sdp->sd_file = ifl;
				sdp->sd_isc = isp;
				sdp->sd_flags = FLG_SY_ISDISC;
				ifl->ifl_oldndx[ndx] = sdp;

				DBG_CALL(Dbg_syms_discarded(ofl->ofl_lml, sdp));
				continue;
			}
		}

		/*
		 * If object capabilities for this file are being converted
		 * into symbol capabilities, then:
		 *
		 *  -	Any global function, or initialized global data symbol
		 *	definitions (ie., those that are not associated with
		 *	special symbol types, ie., ABS, COMMON, etc.), and which
		 *	have not been reduced to locals, are converted to symbol
		 *	references (UNDEF).  This ensures that any reference to
		 *	the original symbol, for example from a relocation, get
		 *	associated to a capabilities family lead symbol, ie., a
		 *	generic instance.
		 *
		 *  -	For each global function, or object symbol definition,
		 *	a new local symbol is created.  The function or object
		 *	is renamed using the capabilities CA_SUNW_ID definition
		 *	(which might have been fabricated for this purpose -
		 *	see get_cap_group()).  The new symbol name is:
		 *
		 *	    <original name>%<capability group identifier>
		 *
		 *	This symbol is associated to the same location, and
		 *	becomes a capabilities family member.
		 */
		/* LINTED */
		hash = (Word)elf_hash(name);

		ntype = ELF_ST_TYPE(nsym->st_info);
		if (cdp && (nsym->st_shndx != SHN_UNDEF) &&
		    ((sdflags & FLG_SY_SPECSEC) == 0) &&
		    ((ntype == STT_FUNC) || (ntype == STT_OBJECT))) {
			/*
			 * Determine this symbol's visibility.  If a mapfile has
			 * indicated this symbol should be local, then there's
			 * no point in transforming this global symbol to a
			 * capabilities symbol.  Otherwise, create a symbol
			 * capability pair descriptor to record this symbol as
			 * a candidate for translation.
			 */
			if (sym_cap_vis(name, hash, sym, ofl) &&
			    ((cpp = alist_append(&cappairs, NULL,
			    sizeof (Cap_pair), AL_CNT_CAP_PAIRS)) == NULL))
				return (S_ERROR);
		}

		if (cpp) {
			Sym	*rsym;

			DBG_CALL(Dbg_syms_cap_convert(ofl, ndx, name, nsym));

			/*
			 * Allocate a new symbol descriptor to represent the
			 * transformed global symbol.  The descriptor points
			 * to the original symbol information (which might
			 * indicate a global or weak visibility).  The symbol
			 * information will be transformed into a local symbol
			 * later, after any weak aliases are culled.
			 */
			if ((cpp->c_osdp =
			    libld_malloc(sizeof (Sym_desc))) == NULL)
				return (S_ERROR);

			cpp->c_osdp->sd_name = name;
			cpp->c_osdp->sd_sym = nsym;
			cpp->c_osdp->sd_shndx = shndx;
			cpp->c_osdp->sd_file = ifl;
			cpp->c_osdp->sd_isc = ifl->ifl_isdesc[shndx];
			cpp->c_osdp->sd_ref = REF_REL_NEED;

			/*
			 * Save the capabilities group this symbol belongs to,
			 * and the original symbol index.
			 */
			cpp->c_group = cdp->ca_groups->apl_data[0];
			cpp->c_ndx = ndx;

			/*
			 * Replace the original symbol definition with a symbol
			 * reference.  Make sure this reference isn't left as a
			 * weak.
			 */
			if ((rsym = libld_malloc(sizeof (Sym))) == NULL)
				return (S_ERROR);

			*rsym = *nsym;

			rsym->st_info = ELF_ST_INFO(STB_GLOBAL, ntype);
			rsym->st_shndx = shndx = SHN_UNDEF;
			rsym->st_value = 0;
			rsym->st_size = 0;

			sdflags |= FLG_SY_CAP;

			nsym = rsym;
		}

		/*
		 * If the symbol does not already exist in the internal symbol
		 * table add it, otherwise resolve the conflict.  If the symbol
		 * from this file is kept, retain its symbol table index for
		 * possible use in associating a global alias.
		 */
		if ((sdp = ld_sym_find(name, hash, &where, ofl)) == NULL) {
			DBG_CALL(Dbg_syms_global(ofl->ofl_lml, ndx, name));
			if ((sdp = ld_sym_enter(name, nsym, hash, ifl, ofl, ndx,
```

```
			    shndx, sdflags, &where)) == (Sym_desc *)S_ERROR)
				return (S_ERROR);

		} else if (ld_sym_resolve(sdp, nsym, ifl, ofl, ndx, shndx,
		    sdflags) == S_ERROR)
			return (S_ERROR);

		/*
		 * Now that we have a symbol descriptor, retain the descriptor
		 * for later use by symbol capabilities processing.
		 */
		if (cpp)
			cpp->c_nsdp = sdp;

		/*
		 * After we've compared a defined symbol in one shared
		 * object, flag the symbol so we don't compare it again.
		 */
		if ((etype == ET_DYN) && (nsym->st_shndx != SHN_UNDEF) &&
		    ((sdp->sd_flags & FLG_SY_SOFOUND) == 0))
			sdp->sd_flags |= FLG_SY_SOFOUND;

		/*
		 * If the symbol is accepted from this file retain the symbol
		 * index for possible use in aliasing.
		 */
		if (sdp->sd_file == ifl)
			sdp->sd_symndx = ndx;

		ifl->ifl_oldndx[ndx] = sdp;

		/*
		 * If we've accepted a register symbol, continue to validate
		 * it.
		 */
		if (sdp->sd_flags & FLG_SY_REGSYM) {
			Sym_desc	*rsdp;

			/*
			 * The presence of FLG_SY_REGSYM means that
			 * the pointers in ld_targ.t_ms are non-NULL.
			 */
			rsdp = (*ld_targ.t_ms.ms_reg_find)(sdp->sd_sym, ofl);
			if (rsdp == NULL) {
				if ((*ld_targ.t_ms.ms_reg_enter)(sdp, ofl) == 0)
					return (S_ERROR);
			} else if (rsdp != sdp) {
				(void) (*ld_targ.t_ms.ms_reg_check)(rsdp,
				    sdp->sd_sym, sdp->sd_name, ifl, ofl);
			}
		}

		/*
		 * For a relocatable object, if this symbol is defined
		 * and has non-zero length and references an address
		 * within an associated section, then check its extents
		 * to make sure the section boundaries encompass it.
		 * If they don't, the ELF file is corrupt. Note that this
		 * global symbol may have come from another file to satisfy
		 * an UNDEF symbol of the same name from this one. In that
		 * case, we don't check it, because it was already checked
		 * as part of its own file.
		 */
		if (etype_rel && (sdp->sd_file == ifl)) {
			Sym *tsym = sdp->sd_sym;
			if (SYM_LOC_BADADDR(sdp, tsym,
			    ELF_ST_TYPE(tsym->st_info))) {
				issue_badaddr_msg(ifl, ofl, sdp,
				    tsym, tsym->st_shndx);
				continue;
			}
		}
	}
	DBG_CALL(Dbg_util_nl(ofl->ofl_lml, DBG_NL_STD));

	/*
	 * Associate weak (alias) symbols to their non-weak counterparts by
	 * scanning the global symbols one more time.
	 *
	 * This association is needed when processing the symbols from a shared
	 * object dependency when a a weak definition satisfies a reference:
	 *
	 *  -	When building a dynamic executable, if a referenced symbol is a
	 *	data item, the symbol data is copied to the executables address
	 *	space.  In this copy-relocation case, we must also reassociate
	 *	the alias symbol with its new location in the executable.
	 *
	 *  -	If the referenced symbol is a function then we may need to
	 *	promote the symbols binding from undefined weak to undefined,
	 *	otherwise the run-time linker will not generate the correct
	 *	relocation error should the symbol not be found.
	 *
	 * Weak alias association is also required when a local dynsym table
	 * is being created.  This table should only contain one instance of a
	 * symbol that is associated to a given address.
	 *
	 * The true association between a weak/strong symbol pair is that both
	 * symbol entries are identical, thus first we create a sorted symbol
	 * list keyed off of the symbols section index and value.  If the symbol
	 * belongs to the same section and has the same value, then the chances
	 * are that the rest of the symbols data is the same.  This list is then
	 * scanned for weak symbols, and if one is found then any strong
	 * association will exist in the entries that follow.  Thus we just have
	 * to scan one (typically a single alias) or more (in the uncommon
	 * instance of multiple weak to strong associations) entries to
	 * determine if a match exists.
	 */
	if (weak && (OFL_ALLOW_LDYNSYM(ofl) || (etype == ET_DYN)) &&
	    (total > local)) {
		static Sym_desc **sort;
		static size_t	osize = 0;
		size_t		nsize = (total - local) * sizeof (Sym_desc *);

		/*
		 * As we might be processing many input files, and many symbols,
		 * try and reuse a static sort buffer.  Note, presently we're
		 * playing the game of never freeing any buffers as there's a
		 * belief this wastes time.
		 */
		if ((osize == 0) || (nsize > osize)) {
			if ((sort = libld_malloc(nsize)) == NULL)
				return (S_ERROR);
			osize = nsize;
		}
		(void) memcpy((void *)sort, &ifl->ifl_oldndx[local], nsize);

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

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

```
			int		sndx;

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

			wsym = wsdp->sd_sym;

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

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

				/*
				 * Ignore any empty symbol descriptor, or the
				 * case where the symbol has been resolved to a
				 * different file.
				 */
				if ((ssdp == NULL) || (ssdp->sd_file != ifl))
					continue;

				ssym = ssdp->sd_sym;

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

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

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

				/*
				 * If a sharable object, set link fields so
				 * that they reference each other.`
				 */
				if (etype == ET_DYN) {
					ssdp->sd_aux->sa_linkndx =
					    (Word)wsdp->sd_symndx;
					wsdp->sd_aux->sa_linkndx =
					    (Word)ssdp->sd_symndx;
				}

				/*
				 * Determine which of these two symbols go into
				 * the sort section.  If a mapfile has made
				 * explicit settings of the FLG_SY_*DYNSORT
```

```
                                 * flags for both symbols, then we do what they
                                 * say.  If one has the DYNSORT flags set, we
                                 * set the NODYNSORT bit in the other.  And if
                                 * neither has an explicit setting, then we
                                 * favor the weak symbol because they usually
                                 * lack the leading underscore.
                                 */
                                w_dynbits = wsdp->sd_flags &
                                    (FLG_SY_DYNSORT | FLG_SY_NODYNSORT);
                                s_dynbits = ssdp->sd_flags &
                                    (FLG_SY_DYNSORT | FLG_SY_NODYNSORT);
                                if (!(w_dynbits && s_dynbits)) {
                                        if (s_dynbits) {
                                                if (s_dynbits == FLG_SY_DYNSORT)
                                                        wsdp->sd_flags |=
                                                            FLG_SY_NODYNSORT;
                                        } else if (w_dynbits !=
                                            FLG_SY_NODYNSORT) {
                                                ssdp->sd_flags |=
                                                    FLG_SY_NODYNSORT;
                                        }
                                }
                                break;
                        }
                }
        }

        /*
         * Having processed all symbols, under -z symbolcap, reprocess any
         * symbols that are being translated from global to locals.  The symbol
         * pair that has been collected defines the original symbol (c_osdp),
         * which will become a local, and the new symbol (c_nsdp), which will
         * become a reference (UNDEF) for the original.
         *
         * Scan these symbol pairs looking for weak symbols, which have non-weak
         * aliases.  There is no need to translate both of these symbols to
         * locals, only the global is necessary.
         */
        if (cappairs) {
                Aliste          idx1;
                Cap_pair        *cpp1;

                for (ALIST_TRAVERSE(cappairs, idx1, cpp1)) {
                        Sym_desc        *sdp1 = cpp1->c_osdp;
                        Sym             *sym1 = sdp1->sd_sym;
                        uchar_t         bind1 = ELF_ST_BIND(sym1->st_info);
                        Aliste          idx2;
                        Cap_pair        *cpp2;

                        /*
                         * If this symbol isn't weak, it's capability member is
                         * retained for the creation of a local symbol.
                         */
                        if (bind1 != STB_WEAK)
                                continue;

                        /*
                         * If this is a weak symbol, traverse the capabilities
                         * list again to determine if a corresponding non-weak
                         * symbol exists.
                         */
                        for (ALIST_TRAVERSE(cappairs, idx2, cpp2)) {
                                Sym_desc        *sdp2 = cpp2->c_osdp;
                                Sym             *sym2 = sdp2->sd_sym;
                                uchar_t         bind2 =
                                    ELF_ST_BIND(sym2->st_info);
```

```

                                if ((cpp1 == cpp2) ||
                                    (cpp1->c_group != cpp2->c_group) ||
                                    (sym1->st_value != sym2->st_value) ||
                                    (bind2 == STB_WEAK))
                                        continue;

                                /*
                                 * The weak symbol (sym1) has a non-weak (sym2)
                                 * counterpart.  There's no point in translating
                                 * both of these equivalent symbols to locals.
                                 * Add this symbol capability alias to the
                                 * capabilities family information, and remove
                                 * the weak symbol.
                                 */
                                if (ld_cap_add_family(ofl, cpp2->c_nsdp,
                                    cpp1->c_nsdp, NULL, NULL) == S_ERROR)
                                        return (S_ERROR);

                                free((void *)cpp1->c_osdp);
                                (void) alist_delete(cappairs, &idx1);
                        }
                }

                DBG_CALL(Dbg_util_nl(ofl->ofl_lml, DBG_NL_STD));

                /*
                 * The capability pairs information now represents all the
                 * global symbols that need transforming to locals.  These
                 * local symbols are renamed using their group identifiers.
                 */
                for (ALIST_TRAVERSE(cappairs, idx1, cpp1)) {
                        Sym_desc        *osdp = cpp1->c_osdp;
                        Objcapset       *capset;
                        size_t          nsize, tsize;
                        const char      *oname;
                        char            *cname, *idstr;
                        Sym             *csym;

                        /*
                         * If the local symbol has not yet been translated
                         * convert it to a local symbol with a name.
                         */
                        if ((osdp->sd_flags & FLG_SY_CAP) != 0)
                                continue;

                        /*
                         * As we're converting object capabilities to symbol
                         * capabilities, obtain the capabilities set for this
                         * object, so as to retrieve the CA_SUNW_ID value.
                         */
                        capset = &cpp1->c_group->cg_set;

                        /*
                         * Create a new name from the existing symbol and the
                         * capabilities group identifier.  Note, the delimiter
                         * between the symbol name and identifier name is hard-
                         * coded here (%), so that we establish a convention
                         * for transformed symbol names.
                         */
                        oname = osdp->sd_name;

                        idstr = capset->oc_id.cs_str;
                        nsize = strlen(oname);
                        tsize = nsize + 1 + strlen(idstr) + 1;
                        if ((cname = libld_malloc(tsize)) == 0)
```

```
                                return (S_ERROR);

                        (void) strcpy(cname, oname);
                        cname[nsize++] = '%';
                        (void) strcpy(&cname[nsize], idstr);

                        /*
                         * Allocate a new symbol table entry, transform this
                         * symbol to a local, and assign the new name.
                         */
                        if ((csym = libld_malloc(sizeof (Sym))) == NULL)
                                return (S_ERROR);

                        *csym = *osdp->sd_sym;
                        csym->st_info = ELF_ST_INFO(STB_LOCAL,
                            ELF_ST_TYPE(osdp->sd_sym->st_info));

                        osdp->sd_name = cname;
                        osdp->sd_sym = csym;
                        osdp->sd_flags = FLG_SY_CAP;

                        /*
                         * Keep track of this new local symbol.  As -z symbolcap
                         * can only be used to create a relocatable object, a
                         * dynamic symbol table can't exist.  Ensure there is
                         * space reserved in the string table.
                         */
                        ofl->ofl_caploclcnt++;
                        if (st_insert(ofl->ofl_strtab, cname) == -1)
                                return (S_ERROR);

                        DBG_CALL(Dbg_syms_cap_local(ofl, cpp1->c_ndx,
                            cname, csym, osdp));

                        /*
                         * Establish this capability pair as a family.
                         */
                        if (ld_cap_add_family(ofl, cpp1->c_nsdp, osdp,
                            cpp1->c_group, &ifl->ifl_caps->ca_syms) == S_ERROR)
                                return (S_ERROR);
                }
        }

        return (1);

#undef SYM_LOC_BADADDR
}

/*
 * Add an undefined symbol to the symbol table.  The reference originates from
 * the location identified by the message id (mid).  These references can
 * originate from command line options such as -e, -u, -initarray, etc.
 * (identified with MSG_INTL(MSG_STR_COMMAND)), or from internally generated
 * TLS relocation references (identified with MSG_INTL(MSG_STR_TLSREL)).
 */
Sym_desc *
ld_sym_add_u(const char *name, Ofl_desc *ofl, Msg mid)
{
        Sym             *sym;
        Ifl_desc        *ifl = NULL, *_ifl;
        Sym_desc        *sdp;
        Word            hash;
        Aliste          idx;
        avl_index_t     where;
        const char      *reference = MSG_INTL(mid);
```

```
        /*
         * As an optimization, determine whether we've already generated this
         * reference.  If the symbol doesn't already exist we'll create it.
         * Or if the symbol does exist from a different source, we'll resolve
         * the conflict.
         */
        /* LINTED */
        hash = (Word)elf_hash(name);
        if ((sdp = ld_sym_find(name, hash, &where, ofl)) != NULL) {
                if ((sdp->sd_sym->st_shndx == SHN_UNDEF) &&
                    (sdp->sd_file->ifl_name == reference))
                        return (sdp);
        }

        /*
         * Determine whether a pseudo input file descriptor exists to represent
         * the command line, as any global symbol needs an input file descriptor
         * during any symbol resolution (refer to map_ifl() which provides a
         * similar method for adding symbols from mapfiles).
         */
        for (APLIST_TRAVERSE(ofl->ofl_objs, idx, _ifl))
                if (strcmp(_ifl->ifl_name, reference) == 0) {
                        ifl = _ifl;
                        break;
                }

        /*
         * If no descriptor exists create one.
         */
        if (ifl == NULL) {
                if ((ifl = libld_calloc(sizeof (Ifl_desc), 1)) == NULL)
                        return ((Sym_desc *)S_ERROR);
                ifl->ifl_name = reference;
                ifl->ifl_flags = FLG_IF_NEEDED | FLG_IF_FILEREF;
                if ((ifl->ifl_ehdr = libld_calloc(sizeof (Ehdr), 1)) == NULL)
                        return ((Sym_desc *)S_ERROR);
                ifl->ifl_ehdr->e_type = ET_REL;

                if (aplist_append(&ofl->ofl_objs, ifl, AL_CNT_OFL_OBJS) == NULL)
                        return ((Sym_desc *)S_ERROR);
        }

        /*
         * Allocate a symbol structure and add it to the global symbol table.
         */
        if ((sym = libld_calloc(sizeof (Sym), 1)) == NULL)
                return ((Sym_desc *)S_ERROR);
        sym->st_info = ELF_ST_INFO(STB_GLOBAL, STT_NOTYPE);
        sym->st_shndx = SHN_UNDEF;

        DBG_CALL(Dbg_syms_process(ofl->ofl_lml, ifl));
        if (sdp == NULL) {
                DBG_CALL(Dbg_syms_global(ofl->ofl_lml, 0, name));
                if ((sdp = ld_sym_enter(name, sym, hash, ifl, ofl, 0, SHN_UNDEF,
                    0, &where)) == (Sym_desc *)S_ERROR)
                        return ((Sym_desc *)S_ERROR);
        } else if (ld_sym_resolve(sdp, sym, ifl, ofl, 0,
            SHN_UNDEF, 0) == S_ERROR)
                return ((Sym_desc *)S_ERROR);

        sdp->sd_flags &= ~FLG_SY_CLEAN;
        sdp->sd_flags |= FLG_SY_CMDREF;

        return (sdp);
}
```

```
/*
 * STT_SECTION symbols have their st_name field set to NULL, and consequently
 * have no name. Generate a name suitable for diagnostic use for such a symbol
 * and store it in the input section descriptor. The resulting name will be
 * of the form:
 *
 *      "XXX (section)"
 *
 * where XXX is the name of the section.
 *
 * entry:
 *      isc - Input section associated with the symbol.
 *      fmt - NULL, or format string to use.
 *
 * exit:
 *      Sets isp->is_sym_name to the allocated string. Returns the
 *      string pointer, or NULL on allocation failure.
 */
const char *
ld_stt_section_sym_name(Is_desc *isp)
{
        const char      *fmt;
        char            *str;
        size_t          len;

        if ((isp == NULL) || (isp->is_name == NULL))
                return (NULL);

        if (isp->is_sym_name == NULL) {
                fmt = (isp->is_flags & FLG_IS_GNSTRMRG) ?
                    MSG_INTL(MSG_STR_SECTION_MSTR) : MSG_INTL(MSG_STR_SECTION);

                len = strlen(fmt) + strlen(isp->is_name) + 1;

                if ((str = libld_malloc(len)) == NULL)
                        return (NULL);
                (void) snprintf(str, len, fmt, isp->is_name);
                isp->is_sym_name = str;
        }

        return (isp->is_sym_name);
}
```