```
**********************************************************
   66541 Wed Mar 27 14:33:40 2013
new/usr/src/cmd/sgs/include/libld.h
3638 ld confuses files with group sections and files that should lazy load
**********************************************************
______unchanged_portion_omitted_

 861 #define FLG_IF_CMDLINE  0x00000001      /* full filename specified from the */
 862                                         /*      command line (no -l) */
 863 #define FLG_IF_NEEDED   0x00000002      /* shared object should be recorded */
 864 #define FLG_IF_DIRECT   0x00000004      /* establish direct bindings to this */
 865                                         /*      object */
 866 #define FLG_IF_EXTRACT  0x00000008      /* file extracted from an archive */
 867 #define FLG_IF_VERNEED  0x00000010      /* version dependency information is */
 868                                         /*      required */
 869 #define FLG_IF_DEPREQD  0x00000020      /* dependency is required to satisfy */
 870                                         /*      symbol references */
 871 #define FLG_IF_NEEDSTR  0x00000040      /* dependency specified by -Nn */
 872                                         /*      flag */
 873 #define FLG_IF_IGNORE   0x00000080      /* ignore unused dependencies */
 874 #define FLG_IF_NODIRECT 0x00000100      /* object contains symbols that */
 875                                         /*      cannot be directly bound to */
 876 #define FLG_IF_LAZYLD   0x00000200      /* dependency should be lazy loaded */
 877 #define FLG_IF_GRPPRM   0x00000400      /* dependency establishes a group */
 878 #define FLG_IF_DISPPEND 0x00000800      /* displacement relocation done */
 879                                         /*      in the ld time. */
 880 #define FLG_IF_DISPDONE 0x00001000      /* displacement relocation done */
 881                                         /*      at the run time */
 882 #define FLG_IF_MAPFILE  0x00002000      /* file is a mapfile */
 883 #define FLG_IF_HSTRTAB  0x00004000      /* file has a string section */
 884 #define FLG_IF_FILEREF  0x00008000      /* file contains a section which */
 885                                         /*      is included in the output */
 886                                         /*      allocatable image */
 887 #define FLG_IF_GNUVER   0x00010000      /* file used GNU-style versioning */
 888 #define FLG_IF_ORDERED  0x00020000      /* ordered section processing */
 889                                         /*      required */
 890 #define FLG_IF_OTOSCAP  0x00040000      /* convert object capabilities to */
 891                                         /*      symbol capabilities */
 892 #define FLG_IF_DEFERRED 0x00080000      /* dependency is deferred */
 893 #define FLG_IF_RTLDINF  0x00100000      /* dependency has DT_SUNW_RTLTINF set */
 894 #define FLG_IF_GROUPS   0x00200000      /* input file has groups to process */
 895 #endif /* ! codereview */

 897 /*
 898  * Symbol states that require the generation of a DT_POSFLAG_1 .dynamic entry.
 899  */
 900 #define MSK_IF_POSFLAG1 (FLG_IF_LAZYLD | FLG_IF_GRPPRM | FLG_IF_DEFERRED)

 902 /*
 903  * Symbol states that require an associated Syminfo entry.
 904  */
 905 #define MSK_IF_SYMINFO  (FLG_IF_LAZYLD | FLG_IF_DIRECT | FLG_IF_DEFERRED)


 908 struct is_desc {                        /* input section descriptor */
 909         const char      *is_name;       /* original section name */
 910         const char      *is_sym_name;   /* NULL, or name string to use for */
 911                                         /*      related STT_SECTION symbols */
 912         Shdr            *is_shdr;       /* the elf section header */
 913         Ifl_desc        *is_file;       /* infile desc for this section */
 914         Os_desc         *is_osdesc;     /* new output section for this */
 915                                         /*      input section */
 916         Elf_Data        *is_indata;     /* input sections raw data */
 917         Is_desc         *is_symshndx;   /* related SHT_SYM_SHNDX section */
 918         Is_desc         *is_comdatkeep; /* If COMDAT section is discarded, */
 919                                         /*      this is section that was kept */
```



```
 920         Word            is_scnndx;      /* original section index in file */
 921         Word            is_ordndx;      /* index for section.  Used to decide */
 922                                         /*      where to insert section when */
 923                                         /*      reordering sections */
 924         Word            is_keyident;    /* key for SHF_{ORDERED|LINK_ORDER} */
 925                                         /*      processing and ident used for */
 926                                         /*       placing/ordering sections */
 927         Word            is_flags;       /* Various flags */
 928 };

 930 #define FLG_IS_ORDERED  0x0001          /* this is a SHF_ORDERED section */
 931 #define FLG_IS_KEY      0x0002          /* section requires sort keys */
 932 #define FLG_IS_DISCARD  0x0004          /* section is to be discarded */
 933 #define FLG_IS_RELUPD   0x0008          /* symbol defined here may have moved */
 934 #define FLG_IS_SECTREF  0x0010          /* section has been referenced */
 935 #define FLG_IS_GDATADEF 0x0020          /* section contains global data sym */
 936 #define FLG_IS_EXTERNAL 0x0040          /* isp from a user file */
 937 #define FLG_IS_INSTRMRG 0x0080          /* Usable SHF_MERGE|SHF_STRINGS sec */
 938 #define FLG_IS_GNSTRMRG 0x0100          /* Generated mergeable string section */

 894 #define FLG_IS_GROUPS   0x0200          /* section has groups to process */
 940 #define FLG_IS_PLACE    0x0400          /* section requires to be placed */
 941 #define FLG_IS_COMDAT   0x0800          /* section is COMDAT */
 942 #define FLG_IS_EHFRAME  0x1000          /* section is .eh_frame */

 944 /*
 945  * Output sections contain lists of input sections that are assigned to them.
 946  * These items fall into 4 categories:
 947  *      BEFORE - Ordered sections that specify SHN_BEFORE, in input order.
 948  *      ORDERED - Ordered sections that are sorted using unsorted sections
 949  *              as the sort key.
 950  *      DEFAULT - Sections that are placed into the output section
 951  *              in input order.
 952  *      AFTER - Ordered sections that specify SHN_AFTER, in input order.
 953  */
 954 #define OS_ISD_BEFORE   0
 955 #define OS_ISD_ORDERED  1
 956 #define OS_ISD_DEFAULT  2
 957 #define OS_ISD_AFTER    3
 958 #define OS_ISD_NUM      4
 959 typedef APlist *os_isdecs_arr[OS_ISD_NUM];

 961 /*
 962  * Convenience macro for traversing every input section associated
 963  * with a given output section. The primary benefit of this macro
 964  * is that it preserves a precious level of code indentation in the
 965  * code that uses it.
 966  */
 967 #define OS_ISDESCS_TRAVERSE(_list_idx, _osp, _idx, _isp) \
 968         for (_list_idx = 0; _list_idx < OS_ISD_NUM; _list_idx++) \
 969                 for (APLIST_TRAVERSE(_osp->os_isdescs[_list_idx], _idx, _isp))


 972 /*
 973  * Map file and output file processing structures
 974  */
 975 struct os_desc {                        /* Output section descriptor */
 976         const char      *os_name;       /* the section name */
 977         Elf_Scn         *os_scn;        /* the elf section descriptor */
 978         Shdr            *os_shdr;       /* the elf section header */
 979         Os_desc         *os_relosdesc;  /* the output relocation section */
 980         APlist          *os_relisdescs; /* reloc input section descriptors */
 981                                         /*      for this output section */
 982         os_isdecs_arr   os_isdescs;     /* lists of input sections in output */
 983         APlist          *os_mstrisdescs; /* FLG_IS_INSTRMRG input sections */
 984         Sg_desc         *os_sgdesc;     /* segment os_desc is placed on */
```

```
 985         Elf_Data        *os_outdata;    /* output sections raw data */
 986         avl_tree_t      *os_comdats;    /* AVL tree of COMDAT input sections */
 987                                         /*      associated to output section */
 988         Word            os_identndx;    /* section identifier for input */
 989                                         /*      section processing, followed */
 990                                         /*      by section symbol index */
 991         Word            os_ordndx;      /* index for section.  Used to decide */
 992                                         /*      where to insert section when */
 993                                         /*      reordering sections */
 994         Xword           os_szoutrels;   /* size of output relocation section */
 995         uint_t          os_namehash;    /* hash on section name */
 996         uchar_t         os_flags;       /* various flags */
 997 };
```

*unchanged_portion_omitted_*

```
**********************************************************
  107298 Wed Mar 27 14:33:41 2013
new/usr/src/cmd/sgs/libld/common/files.c
3638 ld confuses files with group sections and files that should lazy load
**********************************************************
______unchanged_portion_omitted_

2329 /*
2330  * Process a group section.
2331  */
2332 static uintptr_t
2333 process_group(const char *name, Ifl_desc *ifl, Shdr *shdr, Elf_Scn *scn,
2334     Word ndx, int ident, Ofl_desc *ofl)
2335 {
2336         uintptr_t       error;

2338         error = process_section(name, ifl, shdr, scn, ndx, ident, ofl);
2339         if ((error == 0) || (error == S_ERROR))
2340                 return (error);

2342         /*
2343          * Indicate that this input file has groups to process.  Groups are
2344          * processed after all input sections have been processed.
2345          */
2346         ifl->ifl_flags |= FLG_IF_GROUPS;
2346         ifl->ifl_flags |= FLG_IS_GROUPS;

2348         return (1);
2349 }
______unchanged_portion_omitted_

2523 #define MAXNDXSIZE      10

2525 /*
2526  * Process an elf file.  Each section is compared against the section state
2527  * table to determine whether it should be processed (saved), ignored, or
2528  * is invalid for the type of input file being processed.
2529  */
2530 static uintptr_t
2531 process_elf(Ifl_desc *ifl, Elf *elf, Ofl_desc *ofl)
2532 {
2533         Elf_Scn         *scn;
2534         Shdr            *shdr;
2535         Word            ndx, sndx, ordndx = 0, ordcnt = 0;
2536         char            *str, *name;
2537         Word            row, column;
2538         int             ident;
2539         uintptr_t       error;
2540         Is_desc         *vdfisp, *vndisp, *vsyisp, *sifisp;
2541         Is_desc         *capinfoisp, *capisp;
2542         Sdf_desc        *sdf;
2543         Place_path_info path_info_buf, *path_info;

2545         /*
2546          * Path information buffer used by ld_place_section() and related
2547          * routines. This information is used to evaluate entrance criteria
2548          * with non-empty file matching lists (ec_files).
2549          */
2550         path_info = ld_place_path_info_init(ofl, ifl, &path_info_buf);

2552         /*
2553          * First process the .shstrtab section so that later sections can
2554          * reference their name.
2555          */
2556         ld_sup_file(ofl, ifl->ifl_name, elf_kind(elf), ifl->ifl_flags, elf);
```



```
2558         sndx = ifl->ifl_shstrndx;
2559         if ((scn = elf_getscn(elf, (size_t)sndx)) == NULL) {
2560                 ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_GETSCN),
2561                     ifl->ifl_name);
2562                 return (0);
2563         }
2564         if ((shdr = elf_getshdr(scn)) == NULL) {
2565                 ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_GETSHDR),
2566                     ifl->ifl_name);
2567                 return (0);
2568         }
2569         if ((name = elf_strptr(elf, (size_t)sndx, (size_t)shdr->sh_name)) ==
2570             NULL) {
2571                 ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_STRPTR),
2572                     ifl->ifl_name);
2573                 return (0);
2574         }

2576         if (ld_sup_input_section(ofl, ifl, name, &shdr, sndx, scn,
2577             elf) == S_ERROR)
2578                 return (S_ERROR);

2580         /*
2581          * Reset the name since the shdr->sh_name could have been changed as
2582          * part of ld_sup_input_section().
2583          */
2584         if ((name = elf_strptr(elf, (size_t)sndx, (size_t)shdr->sh_name)) ==
2585             NULL) {
2586                 ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_STRPTR),
2587                     ifl->ifl_name);
2588                 return (0);
2589         }

2591         error = process_strtab(name, ifl, shdr, scn, sndx, FALSE, ofl);
2592         if ((error == 0) || (error == S_ERROR))
2593                 return (error);
2594         str = ifl->ifl_isdesc[sndx]->is_indata->d_buf;

2596         /*
2597          * Determine the state table column from the input file type.  Note,
2598          * shared library sections are not added to the output section list.
2599          */
2600         if (ifl->ifl_ehdr->e_type == ET_DYN) {
2601                 column = 1;
2602                 ofl->ofl_soscnt++;
2603                 ident = ld_targ.t_id.id_null;
2604         } else {
2605                 column = 0;
2606                 ofl->ofl_objscnt++;
2607                 ident = ld_targ.t_id.id_unknown;
2608         }

2610         DBG_CALL(Dbg_file_generic(ofl->ofl_lml, ifl));
2611         ndx = 0;
2612         vdfisp = vndisp = vsyisp = sifisp = capinfoisp = capisp = NULL;
2613         scn = NULL;
2614         while (scn = elf_nextscn(elf, scn)) {
2615                 ndx++;

2617                 /*
2618                  * As we've already processed the .shstrtab don't do it again.
2619                  */
2620                 if (ndx == sndx)
2621                         continue;

2623                 if ((shdr = elf_getshdr(scn)) == NULL) {
```

```
2624                            ld_eprintf(ofl, ERR_ELF, MSG_INTL(MSG_ELF_GETSHDR),
2625                                ifl->ifl_name);
2626                            return (0);
2627                    }
2628                    name = str + (size_t)(shdr->sh_name);

2630                    if (ld_sup_input_section(ofl, ifl, name, &shdr, ndx, scn,
2631                        elf) == S_ERROR)
2632                            return (S_ERROR);

2634                    /*
2635                     * Reset the name since the shdr->sh_name could have been
2636                     * changed as part of ld_sup_input_section().
2637                     */
2638                    name = str + (size_t)(shdr->sh_name);

2640                    row = shdr->sh_type;

2642                    /*
2643                     * If the section has the SHF_EXCLUDE flag on, and we're not
2644                     * generating a relocatable object, exclude the section.
2645                     */
2646                    if (((shdr->sh_flags & SHF_EXCLUDE) != 0) &&
2647                        ((ofl->ofl_flags & FLG_OF_RELOBJ) == 0)) {
2648                            if ((error = process_exclude(name, ifl, shdr, scn,
2649                                ndx, ofl)) == S_ERROR)
2650                                    return (S_ERROR);
2651                            if (error == 1)
2652                                    continue;
2653                    }

2655                    /*
2656                     * If this is a standard section type process it via the
2657                     * appropriate action routine.
2658                     */
2659                    if (row < SHT_NUM) {
2660                            if (Initial[row][column] != NULL) {
2661                                    if (Initial[row][column](name, ifl, shdr, scn,
2662                                        ndx, ident, ofl) == S_ERROR)
2663                                            return (S_ERROR);
2664                            }
2665                    } else {
2666                            /*
2667                             * If this section is below SHT_LOSUNW then we don't
2668                             * really know what to do with it, issue a warning
2669                             * message but do the basic section processing anyway.
2670                             */
2671                            if (row < (Word)SHT_LOSUNW) {
2672                                    Conv_inv_buf_t inv_buf;

2674                                    ld_eprintf(ofl, ERR_WARNING,
2675                                        MSG_INTL(MSG_FIL_INVALSEC), ifl->ifl_name,
2676                                        EC_WORD(ndx), name, conv_sec_type(
2677                                        ifl->ifl_ehdr->e_ident[EI_OSABI],
2678                                        ifl->ifl_ehdr->e_machine,
2679                                        shdr->sh_type, 0, &inv_buf));
2680                            }

2682                            /*
2683                             * Handle sections greater than SHT_LOSUNW.
2684                             */
2685                            switch (row) {
2686                            case SHT_SUNW_dof:
2687                                    if (process_section(name, ifl, shdr, scn,
2688                                        ndx, ident, ofl) == S_ERROR)
2689                                            return (S_ERROR);
```



```
2690                                    break;
2691                            case SHT_SUNW_cap:
2692                                    if (process_section(name, ifl, shdr, scn, ndx,
2693                                        ld_targ.t_id.id_null, ofl) == S_ERROR)
2694                                            return (S_ERROR);
2695                                    capisp = ifl->ifl_isdesc[ndx];
2696                                    break;
2697                            case SHT_SUNW_capinfo:
2698                                    if (process_section(name, ifl, shdr, scn, ndx,
2699                                        ld_targ.t_id.id_null, ofl) == S_ERROR)
2700                                            return (S_ERROR);
2701                                    capinfoisp = ifl->ifl_isdesc[ndx];
2702                                    break;
2703                            case SHT_SUNW_DEBUGSTR:
2704                            case SHT_SUNW_DEBUG:
2705                                    if (process_debug(name, ifl, shdr, scn,
2706                                        ndx, ident, ofl) == S_ERROR)
2707                                            return (S_ERROR);
2708                                    break;
2709                            case SHT_SUNW_move:
2710                                    if (process_section(name, ifl, shdr, scn, ndx,
2711                                        ld_targ.t_id.id_null, ofl) == S_ERROR)
2712                                            return (S_ERROR);
2713                                    break;
2714                            case SHT_SUNW_syminfo:
2715                                    if (process_section(name, ifl, shdr, scn, ndx,
2716                                        ld_targ.t_id.id_null, ofl) == S_ERROR)
2717                                            return (S_ERROR);
2718                                    sifisp = ifl->ifl_isdesc[ndx];
2719                                    break;
2720                            case SHT_SUNW_ANNOTATE:
2721                                    if (process_progbits(name, ifl, shdr, scn,
2722                                        ndx, ident, ofl) == S_ERROR)
2723                                            return (S_ERROR);
2724                                    break;
2725                            case SHT_SUNW_COMDAT:
2726                                    if (process_progbits(name, ifl, shdr, scn,
2727                                        ndx, ident, ofl) == S_ERROR)
2728                                            return (S_ERROR);
2729                                    ifl->ifl_isdesc[ndx]->is_flags |= FLG_IS_COMDAT;
2730                                    break;
2731                            case SHT_SUNW_verdef:
2732                                    if (process_section(name, ifl, shdr, scn, ndx,
2733                                        ld_targ.t_id.id_null, ofl) == S_ERROR)
2734                                            return (S_ERROR);
2735                                    vdfisp = ifl->ifl_isdesc[ndx];
2736                                    break;
2737                            case SHT_SUNW_verneed:
2738                                    if (process_section(name, ifl, shdr, scn, ndx,
2739                                        ld_targ.t_id.id_null, ofl) == S_ERROR)
2740                                            return (S_ERROR);
2741                                    vndisp = ifl->ifl_isdesc[ndx];
2742                                    break;
2743                            case SHT_SUNW_versym:
2744                                    if (process_section(name, ifl, shdr, scn, ndx,
2745                                        ld_targ.t_id.id_null, ofl) == S_ERROR)
2746                                            return (S_ERROR);
2747                                    vsyisp = ifl->ifl_isdesc[ndx];
2748                                    break;
2749                            case SHT_SPARC_GOTDATA:
2750                                    /*
2751                                     * SHT_SPARC_GOTDATA (0x70000000) is in the
2752                                     * SHT_LOPROC - SHT_HIPROC range reserved
2753                                     * for processor-specific semantics. It is
2754                                     * only meaningful for sparc targets.
2755                                     */
```

```
2756                                if (ld_targ.t_m.m_mach !=
2757                                    LD_TARG_BYCLASS(EM_SPARC, EM_SPARCV9))
2758                                        goto do_default;
2759                                if (process_section(name, ifl, shdr, scn, ndx,
2760                                    ld_targ.t_id.id_gotdata, ofl) == S_ERROR)
2761                                        return (S_ERROR);
2762                                break;
2763 #if     defined(_ELF64)
2764                        case SHT_AMD64_UNWIND:
2765                                /*
2766                                 * SHT_AMD64_UNWIND (0x70000001) is in the
2767                                 * SHT_LOPROC - SHT_HIPROC range reserved
2768                                 * for processor-specific semantics. It is
2769                                 * only meaningful for amd64 targets.
2770                                 */
2771                                if (ld_targ.t_m.m_mach != EM_AMD64)
2772                                        goto do_default;

2774                                /*
2775                                 * Target is x86, so this really is
2776                                 * SHT_AMD64_UNWIND
2777                                 */
2778                                if (column == 0) {
2779                                        /*
2780                                         * column == ET_REL
2781                                         */
2782                                        if (process_section(name, ifl, shdr,
2783                                            scn, ndx, ld_targ.t_id.id_unwind,
2784                                            ofl) == S_ERROR)
2785                                                return (S_ERROR);
2786                                        ifl->ifl_isdesc[ndx]->is_flags |=
2787                                            FLG_IS_EHFRAME;
2788                                }
2789                                break;
2790 #endif
2791                        default:
2792                        do_default:
2793                                if (process_section(name, ifl, shdr, scn, ndx,
2794                                    ((ident == ld_targ.t_id.id_null) ?
2795                                    ident : ld_targ.t_id.id_user), ofl) ==
2796                                    S_ERROR)
2797                                        return (S_ERROR);
2798                                break;
2799                        }
2800                }
2801        }

2803        /*
2804         * Now that all input sections have been analyzed, and prior to placing
2805         * any input sections to their output sections, process any groups.
2806         * Groups can contribute COMDAT items, which may get discarded as part
2807         * of placement.  In addition, COMDAT names may require transformation
2808         * to indicate different output section placement.
2809         */
2810        if (ifl->ifl_flags & FLG_IF_GROUPS) {
2810        if (ifl->ifl_flags & FLG_IS_GROUPS) {
2811                for (ndx = 1; ndx < ifl->ifl_shnum; ndx++) {
2812                        Is_desc *isp;

2814                        if (((isp = ifl->ifl_isdesc[ndx]) == NULL) ||
2815                            (isp->is_shdr->sh_type != SHT_GROUP))
2816                                continue;

2818                        if (ld_group_process(isp, ofl) == S_ERROR)
2819                                return (S_ERROR);
2820                }
```



```
2821        }

2823        /*
2824         * Now that all of the input sections have been processed, place
2825         * them in the appropriate output sections.
2826         */
2827        for (ndx = 1; ndx < ifl->ifl_shnum; ndx++) {
2828                Is_desc *isp;

2830                if (((isp = ifl->ifl_isdesc[ndx]) == NULL) ||
2831                    ((isp->is_flags & FLG_IS_PLACE) == 0))
2832                        continue;

2834                /*
2835                 * Place all non-ordered sections within their appropriate
2836                 * output section.
2837                 */
2838                if ((isp->is_flags & FLG_IS_ORDERED) == 0) {
2839                        if (ld_place_section(ofl, isp, path_info,
2840                            isp->is_keyident, NULL) == (Os_desc *)S_ERROR)
2841                                return (S_ERROR);
2842                        continue;
2843                }

2845                /*
2846                 * Count the number of ordered sections and retain the first
2847                 * ordered section index. This will be used to optimize the
2848                 * ordered section loop that immediately follows this one.
2849                 */
2850                ordcnt++;
2851                if (ordndx == 0)
2852                        ordndx = ndx;
2853        }

2855        /*
2856         * Having placed all the non-ordered sections, it is now
2857         * safe to place SHF_ORDERED/SHF_LINK_ORDER sections.
2858         */
2859        if (ifl->ifl_flags & FLG_IF_ORDERED) {
2860                for (ndx = ordndx; ndx < ifl->ifl_shnum; ndx++) {
2861                        Is_desc *isp;

2863                        if (((isp = ifl->ifl_isdesc[ndx]) == NULL) ||
2864                            ((isp->is_flags &
2865                            (FLG_IS_PLACE | FLG_IS_ORDERED)) !=
2866                            (FLG_IS_PLACE | FLG_IS_ORDERED)))
2867                                continue;

2869                        /* ld_process_ordered() calls ld_place_section() */
2870                        if (ld_process_ordered(ofl, ifl, path_info, ndx) ==
2871                            S_ERROR)
2872                                return (S_ERROR);

2874                        /* If we've done them all, stop searching */
2875                        if (--ordcnt == 0)
2876                                break;
2877                }
2878        }

2880        /*
2881         * If this is a shared object explicitly specified on the command
2882         * line (as opposed to being a dependency of such an object),
2883         * determine if the user has specified a control definition. This
2884         * descriptor may specify which version definitions can be used
2885         * from this object. It may also update the dependency to USED and
2886         * supply an alternative SONAME.
```

```
2887          */
2888         sdf = NULL;
2889         if (column && (ifl->ifl_flags & FLG_IF_NEEDED)) {
2890                 const char      *base;

2892                 /*
2893                  * Use the basename of the input file (typically this is the
2894                  * compilation environment name, ie. libfoo.so).
2895                  */
2896                 if ((base = strrchr(ifl->ifl_name, '/')) == NULL)
2897                         base = ifl->ifl_name;
2898                 else
2899                         base++;

2901                 if ((sdf = sdf_find(base, ofl->ofl_socntl)) != NULL) {
2902                         sdf->sdf_file = ifl;
2903                         ifl->ifl_sdfdesc = sdf;
2904                 }
2905         }

2907         /*
2908          * Before symbol processing, process any capabilities.  Capabilities
2909          * can reference a string table, which is why this processing is
2910          * carried out after the initial section processing.  Capabilities,
2911          * together with -z symbolcap, can require the conversion of global
2912          * symbols to local symbols.
2913          */
2914         if (capisp && (process_cap(ofl, ifl, capisp) == S_ERROR))
2915                 return (S_ERROR);

2917         /*
2918          * Process any version dependencies.  These will establish shared object
2919          * 'needed' entries in the same manner as will be generated from the
2920          * .dynamic's NEEDED entries.
2921          */
2922         if (vndisp && ((ofl->ofl_flags & (FLG_OF_NOUNDEF | FLG_OF_SYMBOLIC)) ||
2923             OFL_GUIDANCE(ofl, FLG_OFG_NO_DEFS)))
2924                 if (ld_vers_need_process(vndisp, ifl, ofl) == S_ERROR)
2925                         return (S_ERROR);

2927         /*
2928          * Before processing any symbol resolution or relocations process any
2929          * version sections.
2930          */
2931         if (vsyisp)
2932                 (void) ld_vers_sym_process(ofl, vsyisp, ifl);

2934         if (ifl->ifl_versym &&
2935             (vdfisp || (sdf && (sdf->sdf_flags & FLG_SDF_SELECT))))
2936                 if (ld_vers_def_process(vdfisp, ifl, ofl) == S_ERROR)
2937                         return (S_ERROR);

2939         /*
2940          * Having collected the appropriate sections carry out any additional
2941          * processing if necessary.
2942          */
2943         for (ndx = 0; ndx < ifl->ifl_shnum; ndx++) {
2944                 Is_desc *isp;

2946                 if ((isp = ifl->ifl_isdesc[ndx]) == NULL)
2947                         continue;
2948                 row = isp->is_shdr->sh_type;

2950                 if ((isp->is_flags & FLG_IS_DISCARD) == 0)
2951                         ld_sup_section(ofl, isp->is_name, isp->is_shdr, ndx,
2952                             isp->is_indata, elf);
```



```
2954                 /*
2955                  * If this is a SHT_SUNW_move section from a relocatable file,
2956                  * keep track of the section for later processing.
2957                  */
2958                 if ((row == SHT_SUNW_move) && (column == 0)) {
2959                         if (aplist_append(&(ofl->ofl_ismove), isp,
2960                             AL_CNT_OFL_MOVE) == NULL)
2961                                 return (S_ERROR);
2962                 }

2964                 /*
2965                  * If this is a standard section type process it via the
2966                  * appropriate action routine.
2967                  */
2968                 if (row < SHT_NUM) {
2969                         if (Final[row][column] != NULL) {
2970                                 if (Final[row][column](isp, ifl,
2971                                     ofl) == S_ERROR)
2972                                         return (S_ERROR);
2973                         }
2974 #if     defined(_ELF64)
2975                 } else if ((row == SHT_AMD64_UNWIND) && (column == 0)) {
2976                         Os_desc *osp = isp->is_osdesc;

2978                         /*
2979                          * SHT_AMD64_UNWIND (0x70000001) is in the SHT_LOPROC -
2980                          * SHT_HIPROC range reserved for processor-specific
2981                          * semantics, and is only meaningful for amd64 targets.
2982                          *
2983                          * Only process unwind contents from relocatable
2984                          * objects.
2985                          */
2986                         if (osp && (ld_targ.t_m.m_mach == EM_AMD64) &&
2987                             (ld_unwind_register(osp, ofl) == S_ERROR))
2988                                 return (S_ERROR);
2989 #endif
2990                 }
2991         }

2993         /*
2994          * Following symbol processing, if this relocatable object input file
2995          * provides symbol capabilities, tag the associated symbols so that
2996          * the symbols can be re-assigned to the new capabilities symbol
2997          * section that will be created for the output file.
2998          */
2999         if (capinfoisp && (ifl->ifl_ehdr->e_type == ET_REL) &&
3000             (process_capinfo(ofl, ifl, capinfoisp) == S_ERROR))
3001                 return (S_ERROR);

3003         /*
3004          * After processing any symbol resolution, and if this dependency
3005          * indicates it contains symbols that can't be directly bound to,
3006          * set the symbols appropriately.
3007          */
3008         if (sifisp && ((ifl->ifl_flags & (FLG_IF_NEEDED | FLG_IF_NODIRECT)) ==
3009             (FLG_IF_NEEDED | FLG_IF_NODIRECT)))
3010                 (void) ld_sym_nodirect(sifisp, ifl, ofl);

3012         return (1);
3013 }
```

_____***unchanged_portion_omitted_***